Universidad Zaragoza
1542

Trabajo Fin de Grado

Desarrollo y evaluación de una aplicación audiovisual interactiva (videojuego) en red con Unity 3D

Autor

David del Cos Azón

Directores

José Ramón Beltrán Blázquez
José Ruiz Más

Escuela de Ingeniería y Arquitectura

2021

Resumen

Los videojuegos han ganado popularidad en los últimos años, especialmente los videojuegos on-line. La industria de los videojuegos ingresó a nivel mundial 120.100 millones de dólares en 2019, con un videojuego on-line (*Fortnite*) como el que más ingresos generó. Además, desde un punto de vista académico, estos tienen un gran interés por las necesidades de la red (baja latencia y constante, pocas pérdidas de paquetes y cadencia de paquetes constante) para un buen funcionamiento del juego.

En este trabajo de fin de grado se desarrolla un videojuego on-line en una arquitectura cliente-servidor, utilizando herramientas y programas gratuitos (*Unity 3D*, la API de red *Mirror* y *Visual Studio*). Una vez desarrollada la aplicación, se realiza una caracterización del tráfico de red (ancho de banda, paquetes por segundo y tamaño de los paquetes) para los distintos flujos del cliente y servidor con el programa *Wireshark*.

En este tipo de juegos (FPS, *First Person Shooter*) es muy importante la interactividad ya que cuando el usuario realiza una acción necesita que se reproduzca inmediatamente en su pantalla y en la de los otros usuarios. La interactividad puede verse seriamente afectada por condiciones adversas en la red. Por ello, se han realizado test de laboratorio en los que se introducen pérdidas de paquetes, retardos y variaciones de retardo (*jitter*). Variando estos parámetros de calidad de servicio de la red (QoS, *Quality of Service*) se obtiene la calidad de experiencia del usuario (QoE, *Quality of Experience*) mediante un test MOS (*Mean Opinion Score*).

Los resultados muestran que es posible jugar con calidad teniendo un retardo de hasta 200 ms, una pérdida del 20% de paquetes o un *jitter* de entre 100-200 ms. La situación que más afecta a la calidad del juego es la pérdida de paquetes a ráfagas.

Contenido

Lista de acrónimos

FPS	First Person Shooter
QoS	Quality of Service
QoE	Quality of Experience
MOS	Mean Opinion Score
API	Applications Programming Interfaces
RPC	Remote Procedure Call
RTS	Real Time Strategy
MMORPG	Massive Multiplayer Online Role-Playing Game
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VoIP	Voice over IP
RTT	Round-Trip Time
ARQ	Automatic Request Repeat
RTO	Retransmission Time-Out
ACK	Acknowledgement

Lista de figuras

Lista de tablas

1. Introducción

1.1. Introducción y motivación

Los videojuegos on-line han ganado mucha popularidad en los últimos años. Además, tienen unas necesidades de red que resultan interesantes desde un punto de vista académico.

Por ello, la principal motivación es crear un juego en red y evaluarlo utilizando los conocimientos aprendidos durante la carrera. También es una motivación el poder aprender cómo funcionan este tipo de juegos internamente, ver cómo les afectan las condiciones de la red y saber qué equipos se necesitan para desplegar una aplicación de este tipo.

La idea inicial era continuar el videojuego creado en la asignatura *Sistemas Electrónicos de Audio y Video (7º semestre)* y añadirle funcionalidades de red, pero se decidió que era mejor comenzar desde cero y buscar un tipo de juego más interactivo para poder evaluar bien las condiciones de red. Se escogió el género FPS combinado con un juego de plataformas. En este tipo de juegos es muy importante la interactividad ya que cuando un usuario realiza una acción necesita que se reproduzca inmediatamente en su pantalla y en la de los otros usuarios.

1.2. Objetivos

Los objetivos de este trabajo fin de grado son los siguientes:

a) Desarrollar una aplicación audiovisual interactiva (videojuego) en red, utilizando una arquitectura cliente-servidor.

b) Caracterizar el tráfico de red generado por la aplicación (ancho de banda, cadencia y tamaño de los paquetes y flujos).

c) Variar parámetros de calidad de servicio de la red "QoS" (latencia, *jitter* y pérdida de paquetes) y ver cómo afectan a la calidad de experiencia del usuario "QoE" mediante un test MOS (*Mean Opinion Score*). Conocer cuáles son los valores máximos de estos parámetros para tener una calidad aceptable y cuál es el parámetro que más afecta.

1.3. Programas y herramientas

Los programas y herramientas utilizados para el desarrollo del juego son:

- *Unity 3D*: Es un motor de videojuegos, proporciona un entorno 2D o 3D donde se puede desarrollar cualquier aplicación audiovisual.

- *Mirror*: Es una API de red dentro de *Unity* que permite añadir conectividad en red a las aplicaciones. Proporciona RPCs (*Remote Procedure Calls*), comandos y

variables sincronizadas para poder desarrollar las interacciones del juego. Dispone de varios transportes de red a elegir según las necesidades de la aplicación.

- *Microsoft Visual Studio*: Es un editor de código, en este caso se utiliza con C#.

- *Parrel Sync*: Permite clonar el proyecto de *Unity*. Con esta herramienta podemos tener el proyecto abierto en una ventana y un proyecto clon en otra, este clon se va actualizando al cambiar cualquier parte del proyecto original. Esto es útil para comprobar que la aplicación funciona bien utilizando una estructura cliente-servidor en la misma máquina.

- *ProBuilder*: Es una herramienta de *Unity* que acelera la creación de entornos 3D (suelos, puertas, escaleras, cilindros …).

Para el análisis de los parámetros de red usamos:

- *Wireshark*: Se utiliza para capturar tráfico en un adaptador de red. También permite obtener gráficas y estadísticas a partir de las capturas de red.

- *Network Emulator Client*: Con este programa se varían las condiciones de la red. Podemos añadir pérdidas de paquetes, latencia y *jitter* para los flujos de entrada o salida de la tarjeta de red.

1.4. Organización

En este primer capítulo se introduce el tema del trabajo, los objetivos y los programas y herramientas utilizadas.

En el segundo capítulo se revisa el estado del arte de los videojuegos y las aplicaciones disponibles para desarrollarlos, así como las APIs y transportes de red según el tipo de aplicación.

En el tercer capítulo se explica el desarrollo del juego, desde la captura de las entradas del usuario con el teclado y ratón hasta mostrarlo en la pantalla a través de un personaje animado. También se detalla cómo es la estructura de un juego on-line y el proceso de creación de una escena en 3D.

En el cuarto capítulo se caracteriza el tráfico de red de la aplicación. Además, se realizan las pruebas de red a usuarios y su análisis posterior.

En el quinto capítulo, se presentan las conclusiones de todo el trabajo realizado y los problemas encontrados. Por último, se proponen futuras mejoras de este trabajo.

Finalmente, se incluye un anexo con el código desarrollado y se presentan las referencias de este trabajo fin de grado.

2. Estado del arte

En este capítulo se realiza en primer lugar una revisión de la industria de los videojuegos en la actualidad, después se comparan los programas y APIs de red disponibles para crear aplicaciones. Por último, se muestran los transportes de red en función del tipo de aplicación a desarrollar y se justifica la opción elegida.

2.1. La industria de los videojuegos

La industria de los videojuegos ha crecido enormemente en los últimos años, especialmente los videojuegos online. Esta industria facturó 120.100 millones de dólares en 2019, con un videojuego on-line (*Fortnite*) como el que más ingresos generó [6].

La aparición de los motores de videojuegos de manera comercial y la gran cantidad de videotutoriales en *Youtube* han favorecido la creación de videojuegos.

Según la Asociación Española de Videojuegos [2], la facturación total en España fue de 1747 millones de euros en 2020, con un crecimiento del 18% respecto a 2019. Además, está generando 9000 empleos directos y más de 23000 indirectos.

Aunque puede parecer que sólo los adolescentes utilizan este tipo de aplicaciones, la realidad es que un alto porcentaje de las personas adultas también juegan.
Como se puede ver en la figura 2.1, el 26% de las personas entre 45 y 64 años utilizan videojuegos. Los menores de 24 años son los que más juegan, en torno a un 70% de este grupo.

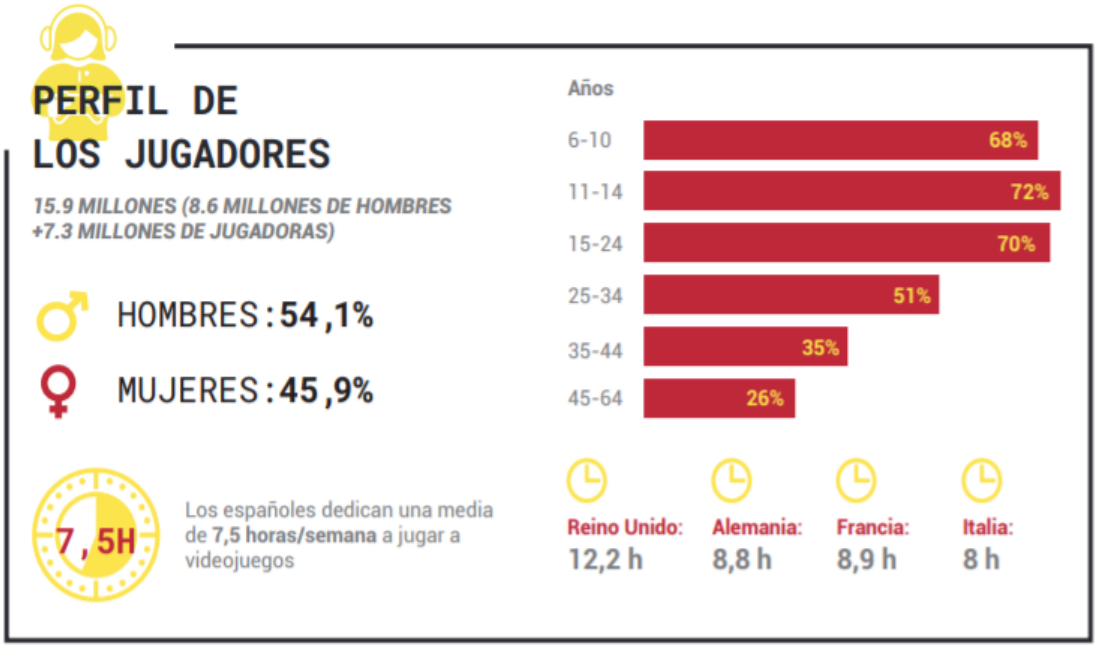


Fig. 2.1 Perfil de los jugadores en España [2]

Algunos estudiantes de la Escuela de Ingeniería y Arquitectura (EINA) también han realizado su trabajo de fin de grado sobre videojuegos en red, principalmente del Grado en Ingeniería Informática. Destacan un videojuego de lucha en red con inteligencia

artificial [3] y otro de carreras en 3D con *Unity* [11]. Ambos realizaron una aplicación completa que permite crear una cuenta de usuario y buscar una lista de servidores disponibles, mientras que este trabajo se ha centrado más en la interactividad y en la calidad de experiencia del usuario.

2.2. Motores de videojuegos

Un motor de videojuegos es "una serie de rutinas de programación que permiten el diseño, la creación y el funcionamiento de un videojuego" [15].

Permite disponer de un espacio de trabajo 2D o 3D (figura 2.2) donde situar objetos y variar su posición. Simula físicas (gravedad, movimiento de partículas, rozamientos de materiales …) y colisiones de estos objetos.

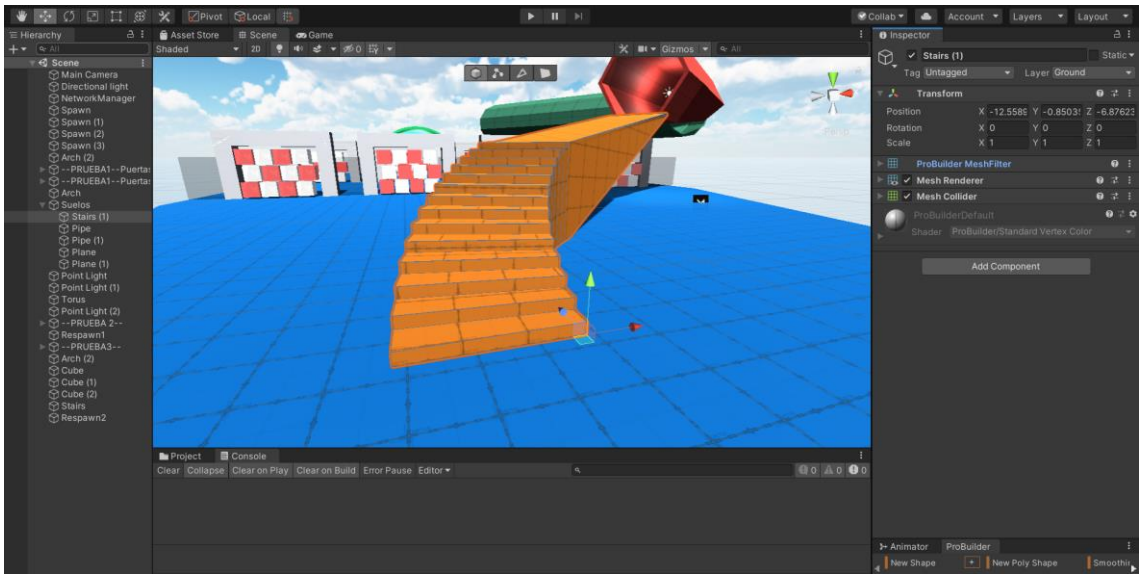


Fig. 2.2 Vista del editor del motor Unity

También proporciona las entradas del usuario de forma sencilla (ratón y teclado) para poder interactuar con el juego. Otros mecanismos de interacción pueden ser una pantalla táctil o un mando de videoconsola. Normalmente, estas entradas del usuario se pueden adaptar para que el juego sea multiplataforma (PC, consola y móvil) sin necesidad de reescribir todo el código, aunque esto depende del motor de videojuegos utilizado.

Los principales motores de videojuegos 3D son *Unity* [17], *Unreal Engine* [23] y *CryEngine* [5]. Estos disponen de varias versiones según las necesidades.

En la tabla 2.1 se realiza una comparativa de las características principales de estos motores.

Tabla 2.1 Comparativa de motores de videojuegos

	Coste	**Ganancias**	**Calidad**	**Dificultad**	**Lenguaje**	**Juegos**	**Plata-formas**
Unreal Engine	5% de las ganancias o gratis	Gratuito hasta 3000$ por trimestre	Media/Alta	Media/Alta	C++	Fortnite	PC Consola
Unity Personal	Gratis	Hasta 100000$/año	Media	Baja/Media	C#	Rust, Fall Guys	PC Consola Móvil
Unity Plus	400$/año	Hasta 200000$/año	-	-	-	-	-
Unity Pro	1800$/año	Sin límite	-	-	-	-	-
Cry Engine	5% de las ganancias o gratis	Gratuito hasta 5000$/año	Alta	Alta	Lua	Farcry	PC Consola Móvil

Para el desarrollo del juego se escoge el motor *Unity* por su sencillez y por ser gratuito hasta 100000$ anuales de ganancias. Muchas empresas grandes desarrollan sus propios motores para evitar pagar estas tasas.
Aunque la calidad de *Unity* es algo menor que la de otros motores, es suficiente para el objetivo de este trabajo. Destaca el uso del lenguaje C#, que es más sencillo que C++, y la posibilidad de crear aplicaciones multiplataforma de manera fácil.

2.3. APIs de red

Una vez elegido el motor de videojuegos, es necesaria una manera de interconectar a los diferentes usuarios en red.
Para agilizar el proceso se utiliza una API de red (*Applications Progamming Interfaces*). Las APIs de red disponibles dentro de Unity son las siguientes:

- *UNET*: Es la API propietaria de *Unity*, pero está obsoleta como aparece en la página oficial [22].

- *Photon Unity Networking*: Tiene una versión gratuita (*PUN Free*) y otra de pago (*PUN Plus*) [9].

- *Mirror*: Es una API de red gratuita [14].

Entre estas opciones, se descarta *UNET* ya que está obsoleta. En la tabla 2.2 se realiza una comparativa de las dos opciones restantes.

Tabla 2.2 Comparativa de APIs de red para Unity

	Coste por año	**Licencia**	**Nº máximo de usuarios concurrentes (CCU)**	**Servidores**
Mirror	0$	Open Source	Sin límite	Cualquier PC
Photon Free	0$	Demo	20 CCU	Photon Cloud
Photon Plus	95-4440	Licencia Mensual	100 CCU-2000 CCU	Photon Cloud

En la tabla 2.2 se puede ver que la mejor opción es *Mirror*. Esta API de red es gratuita, de código abierto y permite ejecutar un servidor en cualquier máquina, mientras que *Photon* está orientado a usar sus servidores propios y a pagar una licencia de uso por ellos.

Además, tras realizar una prueba de ambas opciones, se comprobó que *Mirror* es mucho más sencillo de utilizar ya que contiene algunos ejemplos de juegos básicos (tanques, ping-pong y chat de mensajes).

En este trabajo se utiliza como base el ejemplo de un juego de tanques que nos proporciona la API *Mirror*.

2.4. Tipos de juegos

Antes de elegir el transporte de red hay que tener claro qué tipo de aplicación se va a desarrollar. No es lo mismo una aplicación en la que el usuario interactúa una vez cada varios segundos como en los juegos de estrategia en tiempo real (RTS, *Real Time Strategy*) o los juegos de rol masivos (MMORPG, *Massive Multiplayer Online Role Playing Game*) que un juego donde los usuarios interactúan continuamente, actualizando su posición y realizando diversas acciones, muchas veces por segundo (juegos FPS y de plataformas). Por tanto, la interactividad determina el protocolo de transporte de red a utilizar: TCP o UDP.

En este trabajo se realiza un juego de tipo FPS combinado con un juego de plataformas. La idea es que el usuario tenga que interactuar continuamente con la aplicación y que necesite ver las acciones de otros usuarios rápidamente para poder realizar sus movimientos. Este tipo de juegos tienen importantes restricciones temporales, ya que al ser tan interactivos necesitan una cadencia de imágenes alta para tener una buena experiencia de usuario. A nivel de red esto se traduce en que tienen que enviar paquetes pequeños continuamente con una cierta cadencia, una situación similar a una llamada de VoIP o una videollamada. Además, los paquetes no deben retrasarse ya que el usuario los necesita 'inmediatamente'. Y todo esto no siempre es posible a consecuencia de las variaciones de retardo que sufren los paquetes en la red, por lo que es necesario aplicar un buffer en recepción para que los paquetes puedan ser reproducidos con una cadencia

constante. Con estas características queda claro que el protocolo a usar debe ser UDP frente a un TCP que penaliza la interactividad con sus funciones de control de errores y de congestión.

Hemos de destacar, asimismo, que los juegos FPS necesitan poco ancho de banda. Ello se debe principalmente a una razón histórica, ya que cuando se crearon estos juegos on-line el ancho de banda en los hogares era mucho más limitado que hoy en día (comunicaciones vía modem telefónico). Otras razones se deben a la escalabilidad de la arquitectura cliente-servidor, ya que algunos juegos llegan a tener cientos de usuarios simultáneos y demandan un ancho de banda consumido por usuario lo más bajo posible. Ejemplo de ello es el juego FPS denominado *Counter Strike* del año 2000. El ancho de banda de un modem era de hasta 33 Kbps frente a los 100 Mbps de una conexión de fibra actual.

2.5. *Transporte de red*

La API de red *Mirror* indica que se pueden utilizar una gran variedad de transportes de red, basados en UDP y TCP, según el tipo de aplicación dónde se va a ejecutar. En la literatura existente podemos encontrar una comparativa entre aquellos que nos ofrece la API de red y que nosotros podemos utilizar para el desarrollo de nuestro juego. Así en [8] encontramos una comparativa entre KCP (UDP), TCP y *RakNet* (UDP). En la prueba realizada, el cliente envía 50 bytes 60 veces por segundo, el servidor los devuelve y se calcula el tiempo de ida y vuelta del paquete (RTT, *Round-Trip Time*). El RTT es el intervalo de tiempo desde que el cliente envía el paquete hasta que lo recibe de vuelta del servidor. También se introduce una probabilidad de pérdida de paquetes del 0.5%. Todo esto durante 5 minutos para cada uno de los transportes. El resultado es la figura 2.3, donde se representa el percentil de paquetes con un tiempo de ida y vuelta determinado.

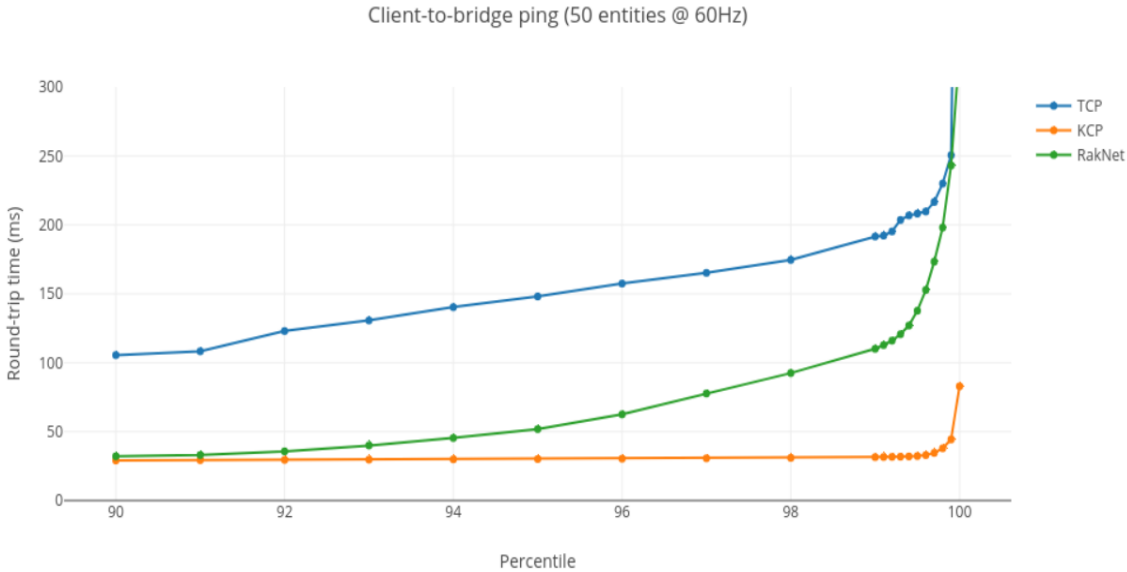


Fig. 2.3 Percentil de paquetes con un RTT determinado para los transportes KCP, TCP y *RakNet*[10]

El tiempo de ida y vuelta se mantiene por debajo de 50 ms en un mayor porcentaje de los paquetes con en el transporte KCP (99.8% de los paquetes). Con *RakNet* el 94% de los paquetes tienen un RTT por debajo de 50 ms. Utilizando TCP el RTT mínimo es de 100

ms y llega hasta 250 ms como máximo. Se concluye que la mejor opción, si se necesita un RTT bajo y estable, es KCP.

En [24] se comparan los transportes KCP, *Enet* y UDT, todos ellos basados en UDP. El objetivo de esta comparativa es saber cuál es la mejor opción para juegos FPS. El cliente envía 500 bytes cada 50 ms y el servidor los envía de vuelta tras recibirlos. El cliente está conectado por wifi y utiliza ADSL, con un ancho de banda de 10 Mb/s. El servidor está en otra localización con un ancho de banda de 5 Mb/s.

Las conclusiones de esta prueba son:

- UDT no es adecuado para un juego FPS, no se comporta bien en situaciones con latencia elevada.
- *Enet* se comporta mejor que UDT cuando aparece latencia en la red.
- KCP es 3 veces mejor que *Enet* en situaciones con latencia elevada.
- KCP es la mejor opción para juegos FPS.

Finalmente, en [20] se compara KCP (UDP), KCP Turbo (UDP), TCP y *Enet* (UDP). En esta comparativa hay dos objetivos: saber cuál es el mejor transporte para juegos FPS y ver cuál se comporta mejor ante la pérdida de paquetes. Para ello se realizan dos pruebas diferentes. En ellas el cliente envía 400 bytes 60 veces por segundo, el servidor los devuelve y se calcula el RTT (Fig. 2.4). Las pruebas son las siguientes:

- Prueba en condiciones normales (sin pérdidas) con BW = 1.05Mb/s y RTT = 27.48 ms con una desviación de 9.74 ms.

- Prueba con una pérdida de paquetes del 10 %.

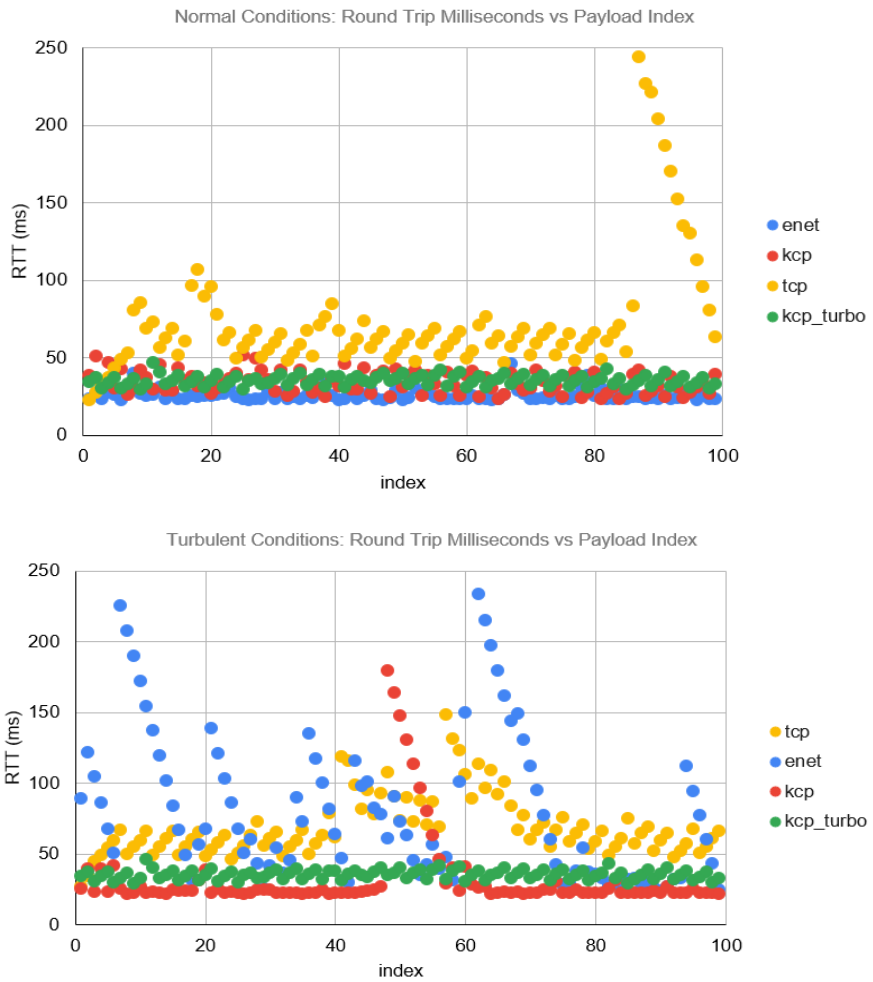


Fig. 2.4 Tiempo de ida y vuelta del paquete en condiciones normales (arriba) y con pérdida de paquetes del 10% (abajo) [20].

En condiciones normales, *Enet* tiene el tiempo de ida y vuelta más bajo, después está KCP con un RTT más variable y que llega hasta 50 ms. Se puede ver que TCP tiene un RTT mucho más elevado y variable, por lo que este transporte no es adecuado para este tipo de aplicaciones. Con una pérdida de paquetes del 10%, KCP ofrece un buen rendimiento, mientras que *Enet* tiene un tiempo de ida y vuelta muy variable.

Tras el análisis de las comparativas presenciales, se ha optado por un transporte KCP para la aplicación. Este protocolo, basado en UDP, incorpora un control de errores ARQ *Selective Repeat* y permite la utilización de corrección de errores FEC. Según la documentación de KCP [10] reduce la latencia entre un 30 y 40% y el máximo retardo en un factor 3 respecto a TCP. Sus características de control de errores son las siguientes:

- Solo retransmite los paquetes que se han perdido.
- Retransmisión rápida: Por ejemplo, si el usuario envía los paquetes 1, 2, 3, 4 y 5, y recibe los ACK 1, 3, 4 y 5. Una vez que recibe el ACK del paquete 4 sin haber recibido el ACK del 2, KCP retransmite el paquete 2 inmediatamente.
- El cálculo del timeout es 1.5xRTO (*Retransmission Timeout*) frente a 2xRTO de TCP.

3. Desarrollo del juego

En este capítulo se explica el desarrollo del juego. En primer lugar, se introducen las características de los objetos en Unity y cómo se pueden mover en un espacio 3D. Después, se muestra la interacción del usuario con el juego y cómo se presenta en la pantalla a través de un personaje animado. A continuación, se detalla la estructura de un juego on-line y se muestran las herramientas que ofrece *Mirror* para realizar las interacciones en red. Finalmente, se introduce brevemente la creación de una escena en 3D y la temática del juego.

3.1. Introducción a Unity

En *Unity* se caracteriza la situación de un objeto en un espacio tridimensional con su propiedad *Transform*, donde se almacenan su posición, rotación y escala. Los valores son de tipo float. Se puede ver en el siguiente ejemplo con un cubo (Fig.3.1).

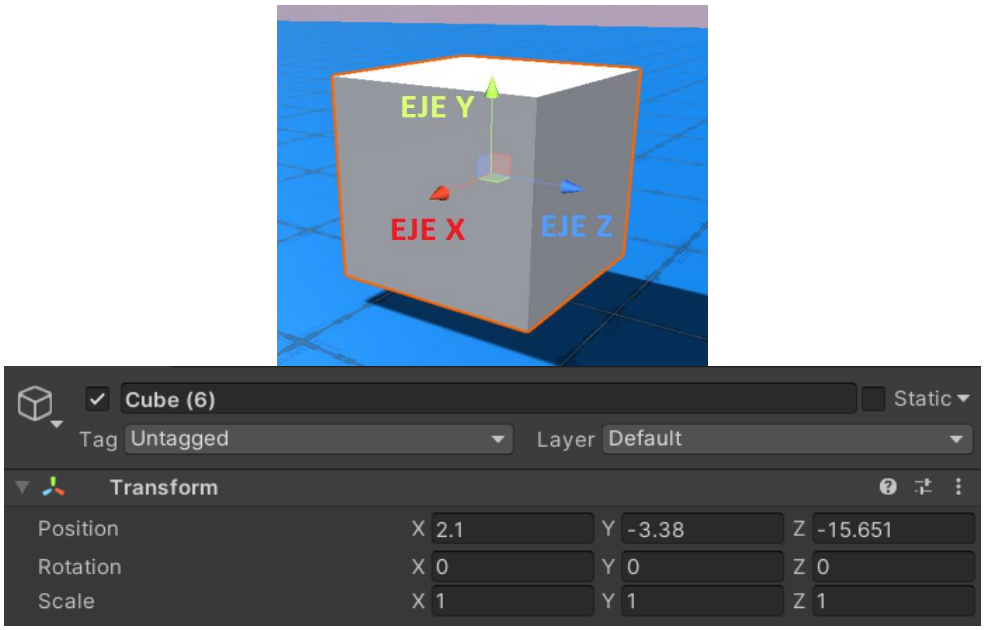


Fig. 3.1 Coordenadas de un objeto en *Unity*

Los movimientos básicos que se pueden realizar son:

- Traslación: Variar las coordenadas de un objeto para hacer que se mueva. Se puede variar la posición de un objeto respecto al tiempo con la instrucción *transform.Translate(mov.x,mov.y,mov.z)*, donde 'mov' es un vector que almacena la variación de la posición en los tres ejes. Antes de aplicarlo, se multiplica el vector por la variación de tiempo respecto al *frame* anterior (*Time.deltaTime*).

- Rotación: Para rotar un objeto se utiliza la instrucción *transform.rotation = Quaternion.Euler(rot.x,rot.y,rot.z)*. Al igual que en el caso anterior, 'rot' es un vector de 3 posiciones (x,y,z), en este caso almacena la rotación de los 3 ejes. Se utilizan los cuaterniones (espacio vectorial de dimensión 4) para aplicar el giro, ya que si se usan los ángulos de Euler puede dar lugar al

bloqueo del cardán (*gimbal lock*), es decir, se pierde un grado de libertad en el giro [4].

3.2. Interfaz de usuario y movimientos

El juego está orientado a PC, por lo que el usuario controlará el juego con el teclado y el ratón. También es posible adaptarlo a un mando de *PlayStation* o *Xbox* ya que *Unity* es compatible con otras plataformas.

Hay que tener en cuenta que es un juego on-line, la posición no se actualiza como en un juego local. Para ello, primero se va a explicar cómo se controla el jugador en un juego local.

Para el control del personaje nos basamos en el tutorial [19] y para realizar las animaciones nos basamos en [16]. Ambos son tutoriales muy básicos para juegos locales. El tutorial de las animaciones solo cubre las animaciones de reposo y movimientos hacia delante y hacia atrás. El tutorial de control del personaje se ha utilizado para simular la gravedad del salto ya que al ser un juego en red no se pueden utilizar directamente las físicas que proporciona *Unity*.

La lista de acciones que se pueden realizar en el juego es:

- Movimientos del personaje hacia delante, hacia atrás y laterales.
- Movimiento de salto del personaje.
- Movimiento de rotación del personaje y de la cámara.
- Apuntar y disparar flechas con el arco

En los siguientes subapartados se presentan los movimientos. Las acciones de apuntar y disparar se explican en los apartados 3.4 y 3.5.

3.2.1. Movimientos hacia delante, hacia atrás y laterales

El movimiento del personaje se controla con las teclas A, W, S y D. Obtenemos las entradas con *GetKey* y actualizamos la posición respecto al tiempo con *transform.Translate*.

```
if (Input.GetKey(KeyCode.W)) { movement.z += 1; }
if (Input.GetKey(KeyCode.S)) { movement.z -= 1; }
if (Input.GetKey(KeyCode.A)) { movement.x -= 1; }
if (Input.GetKey(KeyCode.D)) { movement.x += 1; }

transform.Translate(movement*Time.deltaTime*Speed, Space.Self);
```

3.2.2. Movimiento de salto

El salto se controla con la barra espaciadora. Se comprueba si el jugador está tocando el suelo (*isGrounded*) y pulsando la barra espaciadora a la vez. Si es así, se aplica una velocidad en el eje y correspondiente a la altura h que queremos alcanzar con el salto mediante la siguiente fórmula:

$$v = \sqrt{h * 2 * g} \qquad \textbf{(Ec. 3.1)}$$

La gravedad se aplica como una variación de posición respecto al tiempo en el eje y.

$$\Delta y = \frac{1}{2} \cdot g \cdot t^2 \qquad \textbf{(Ec. 3.2)}$$

El código para aplicar la gravedad al personaje es el siguiente:

```
velocidadCaida.y += gravedad * Time.deltaTime;
controller.Move(velocidadCaida * Time.deltaTime);// (Ecuación) variación posición = 1/2*g*t^2
```

Para comprobar que el jugador toca el suelo, se coloca una esfera en sus pies y se calcula la distancia de esta esfera al suelo. Si la distancia 'd' es menor que 0.4f, se asigna el valor verdadero a la variable booleana *isGrounded*. Esto activa el booleano "Suelo" para reproducir la animación de aterrizaje.

```
isGrounded = Physics.CheckSphere(sueloCheck.position, distanciaSuelo, sueloMask);
animator.SetBool("Suelo", Physics.CheckSphere(sueloCheck.position, 1.0f, sueloMask));
```

En la figura 3.2 se puede ver el movimiento de salto.

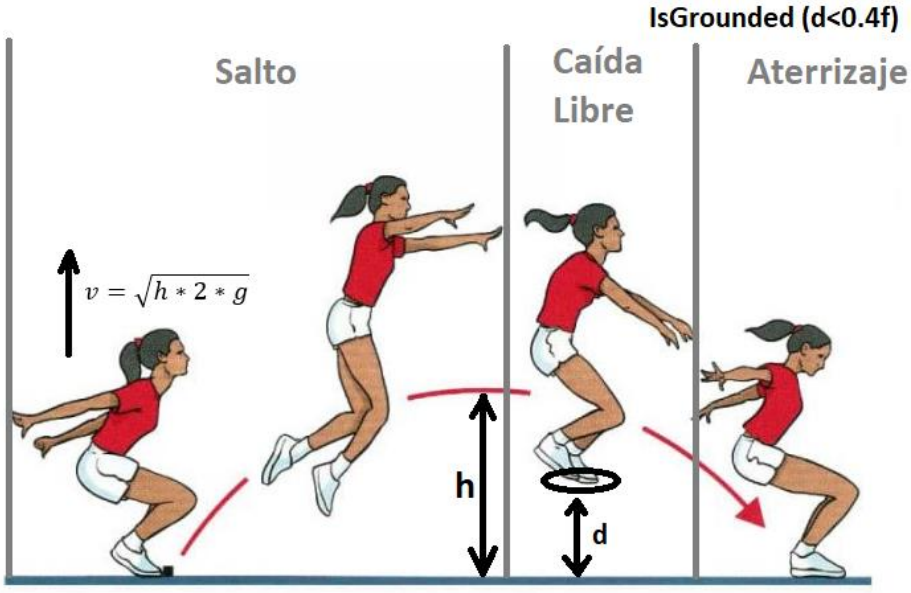


Fig. 3.2 Salto del personaje

3.2.3. Movimientos de rotación del personaje y de la cámara

La rotación y la vista del personaje se controlan con el ratón. Se utilizan las entradas *Mouse X* y *Mouse Y* que proporciona *Unity* (Fig.3.3).

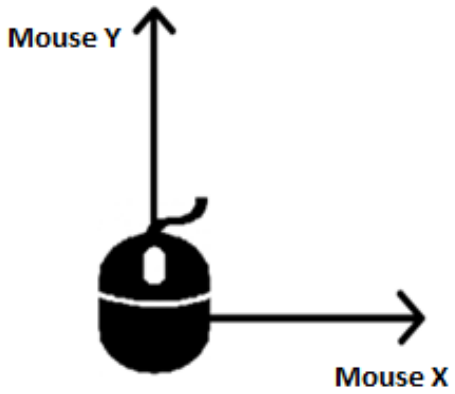


Fig. 3.3 Entradas del ratón

Aunque el análisis de red será como el de un juego en primera persona, la cámara es en tercera persona. Rotaremos la cámara en el eje x e y con 'Mouse X' y 'Mouse Y' respectivamente, mientras que el personaje solo rotara en el eje horizontal con 'Mouse X'.

```
mouseX += Input.GetAxis("Mouse X")*RotationSpeed;//Eje horizontal
mouseY -= Input.GetAxis("Mouse Y")*RotationSpeed;//Eje vertical (vista)
mouseY = Mathf.Clamp(mouseY, -20, 60)

transform.rotation = Quaternion.Euler(0, mouseX, 0);
Camera.main.transform.rotation = Quaternion.Euler(mouseY, mouseX, 0);
```

En la figura 3.4 se pueden ver las rotaciones del personaje y la posición de la cámara.

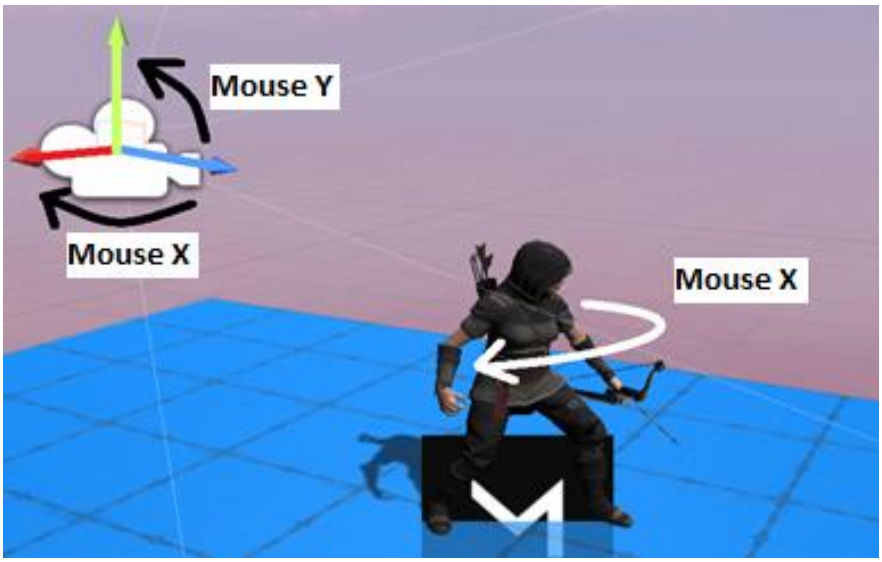


Fig. 3.4 Rotación de la cámara y del personaje

3.3. Animaciones

Para conseguir una mayor interacción se utiliza un personaje con animaciones. Según las entradas del usuario (teclado y ratón), se reproduce una animación u otra. Todas las animaciones se controlan con una máquina de estados de animaciones
El personaje y las animaciones se obtienen en la página web *Mixamo* [1]. *Mixamo* proporciona una gran variedad de modelos 3D de alta calidad con animaciones realizadas por actores. Todas estas herramientas son libres de derechos de autor y pueden ser utilizadas en proyectos personales, sin ánimo de lucro y comerciales.

3.3.1. Importar un personaje

Primero se descarga el modelo 3D del personaje en formato *FBX for Unity (Fig*. 3.5).

Fig. 3.5 Personaje 3D de *Mixamo*

El juego se basa en este personaje que lleva un arco y puede disparar flechas. Las flechas son un objeto diferente, en el apartado 3.5 se explica cómo se realizan los disparos. Una vez obtenido el modelo, se descargan las animaciones. Estas tienen 30 fotogramas por segundo.

3.3.2. Animaciones básicas: reposo y movimiento

Para animar un personaje es necesaria una máquina de estados. En ella se requieren al menos 2 estados: IDLE (animación de reposo) y movimiento (animación de movimiento hacia delante, hacia atrás y lateral). Dentro del estado movimiento, según las entradas del usuario (botones A, W, S o D), se reproduce una animación u otra. El cambio entre estos dos estados se realiza con el booleano ‘Caminando’.

3.3.3. Animación de salto

El salto se compone de 3 estados: salto, caída libre y aterrizaje. La animación de salto se recorta en 3 partes y se incorporan a cada uno de estos 3 estados. El salto comienza cuando el booleano ‘Salta’ es cierto (el usuario pulsa la barra espaciadora del teclado y la variable *isGrounded* es cierta). Para comprobar que el personaje ha tocado el suelo se usa el booleano ‘Suelo’, que actualizamos desde el código como se ha visto anteriormente, cuando es cierto se pasa de caída libre a aterrizaje. Además, hay otra animación de salto en movimiento.

```
if (Input.GetButtonDown("Jump") && isGrounded && animator.GetCurrentAnimatorStateInfo(0).IsName("IDLE
(Reposo)"))
        {
            animator.SetBool("Salta", true);
            velocidadCaida.y = Mathf.Sqrt(fuerzaSalto * -2f * gravedad);// (Ecuación) v = sqrt(h*2*g)

        }
        else if (Input.GetButtonDown("Jump") && isGrounded &&
animator.GetCurrentAnimatorStateInfo(0).IsName("Movimiento"))
        {
            animator.SetBool("SaltaCorriendo", true);
            velocidadCaida.y = Mathf.Sqrt(fuerzaSalto * -2f * gravedad);// (Ecuación) v = sqrt(h*2*g)
        }
        else
        {
            animator.SetBool("Salta", false);
            animator.SetBool("SaltaCorriendo", false);
        }
}
```

3.3.4. Animación de apuntado y disparo

Las animaciones de apuntado con el arco (en reposo y movimiento) se controlan con el booleano ‘Apuntando’ (es cierto cuando el usuario mantiene el botón derecho del ratón). Finalmente, se incorpora la animación de disparo. Es posible disparar desde cualquier estado. Esta animación se controla con la variable ‘Dispara’ de tipo *trigger* (es cierta cuando el usuario pulsa el botón izquierdo del ratón y vuelve a ser falsa tras ejecutarse la animación). La animación de apuntado se realiza con el siguiente código:

```
apunta = false;
if (Input.GetMouseButton(1))
{
        apunta = true;
        Velocidad = 4.0f;
}
else
{

        Velocidad = 8.0f;
}

animator.SetBool("Apuntando", apunta);
```

3.3.5. Máquina de estados de animaciones

En la figura 3.6 se puede ver la maquina de estados final.

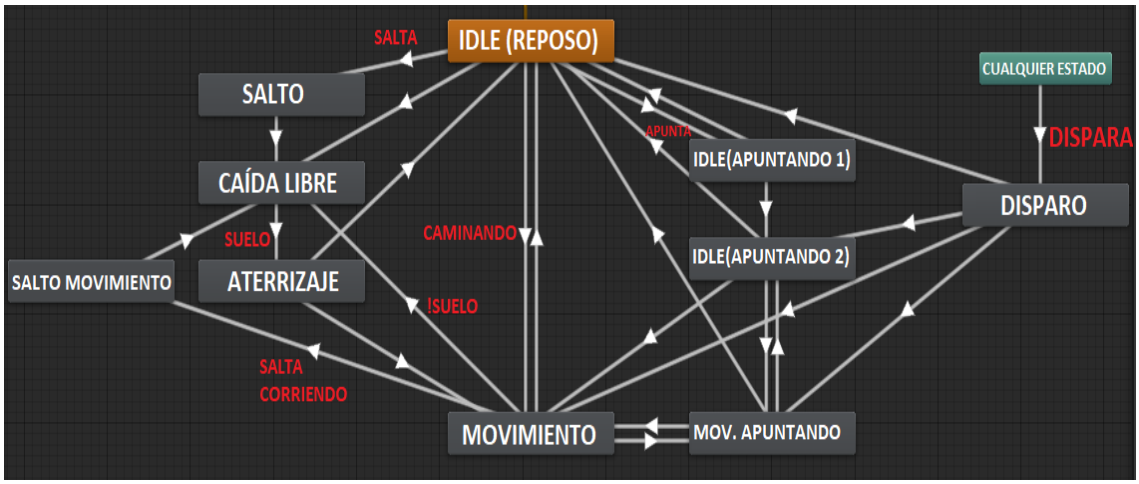


Fig. 3.6 Máquina de estados de animaciones

3.4. Estructura de la aplicación

Esta aplicación utiliza una arquitectura cliente-servidor. La aplicación se gestiona desde el bloque *NetworkManager*. Este bloque permite escoger el transporte de red de la aplicación (KCP) y el puerto donde escucha el servidor (7777 en este caso). Los objetos que pueden ser creados en la partida (como personajes o las flechas) se añaden en la opción *Player Object*. También permite seleccionar el número máximo de clientes por partida y la tasa de refresco del servidor (60 veces por segundo). En la figura 3.7 se puede visualizar el bloque *NetworkManager* con todas las características que se han explicado anteriormente.

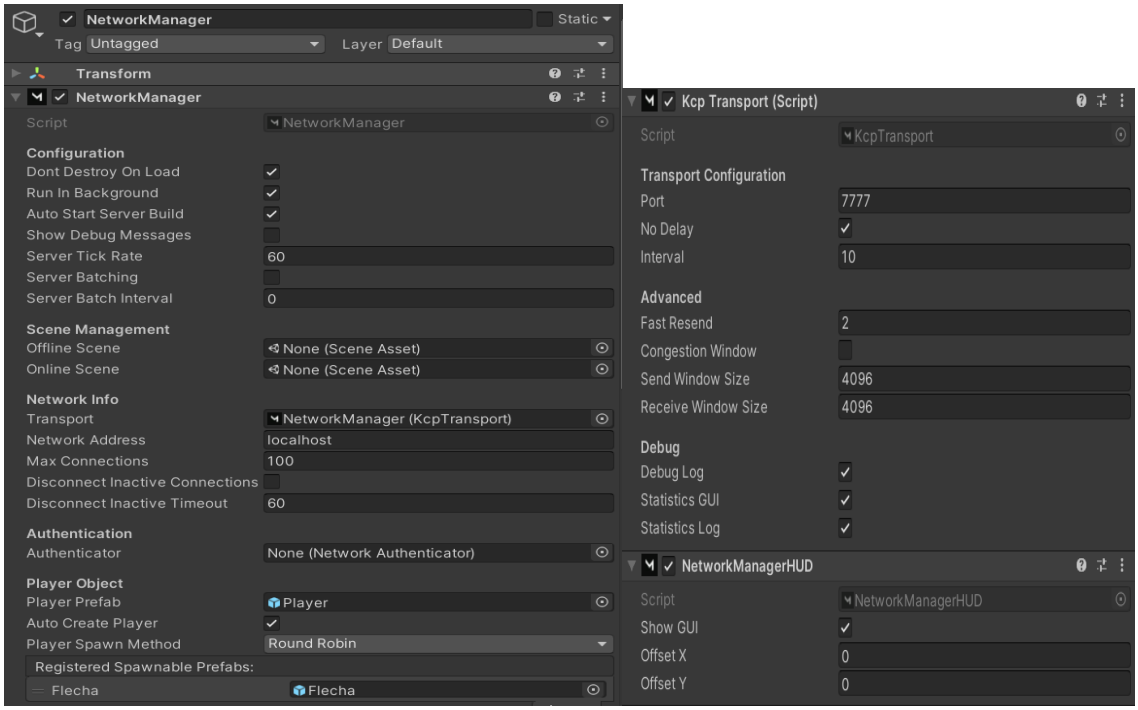


Fig. 3.7 Bloque *NetworkManager*

Además, tiene una interfaz gráfica visible en el juego (*NetworkManagerHUD*) que permite iniciar partidas como servidor o unirse a otras como cliente (Fig. 3.8). Para unirse como cliente hay que introducir la dirección IP del servidor en el cuadro que aparece localhost. El puerto del servidor es fijo, mientras que el puerto del cliente es aleatorio.

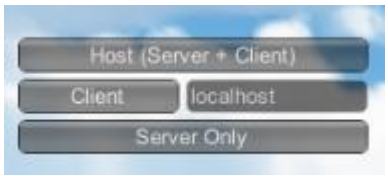


Fig. 3.8 Interfaz gráfica de usuario (*NetworkManagerHUD*)

Por otro lado, en vez de tener códigos separados para el servidor y el cliente, el código de un objeto (*Network Behaviour*) se comparte entre ambos. Por ello, la API de red proporciona herramientas para ejecutar partes de este código solo en el servidor o en el cliente.

Esto se realiza colocando las siguientes etiquetas encima de una función o código para que solo le afecte a este.

- [*Server*]: El código que se encuentre tras la etiqueta solo se ejecuta en el servidor.
- [*Client*]: En este caso solo se ejecuta en el cliente.

Por ejemplo, si se necesita que la función prueba( ) se ejecute solo en los clientes, se utiliza el siguiente código:

```
[Client]
void prueba() {} //Solo lo ejecutan los clientes
```

También hay otras etiquetas para realizar las interacciones en red:

- [*ClientRpc*]: El servidor utiliza está llamada a procedimiento remoto o RPC (*Remote Call Procedure*) para que una función se ejecute en todos los clientes.
- [*TargetRpc*]: Igual que la etiqueta anterior, pero en este caso la función se ejecuta en un solo cliente.
- [*Command*]: Los clientes pueden usar un comando para ejecutar una función en el servidor.
- [*SyncVar*]: Se puede utilizar para sincronizar variables automáticamente. Se coloca encima de la variable y cada vez que la variable cambia en el servidor, se actualiza en todos los clientes. En la aplicación no se ha utilizado este tipo de etiqueta.

Además, *Mirror* permite comprobar quién tiene autoridad sobre un objeto o personaje para que solo el propietario pueda controlar el objeto. Para ello proporciona las siguientes variables booleanas.

- *isServer*: Devuelve verdadero si lo está ejecutando el servidor.
- *isClient*: Similar al anterior, pero para el cliente.
- *isLocalPlayer*: Verdadero si el cliente ha creado este objeto.
- *hasAuthority*: Verdadero si el cliente tiene autoridad sobre el objeto, es decir, si puede controlarlo.

En la figura 3.9 se pueden ver las interacciones del juego utilizando comandos y RPCs.

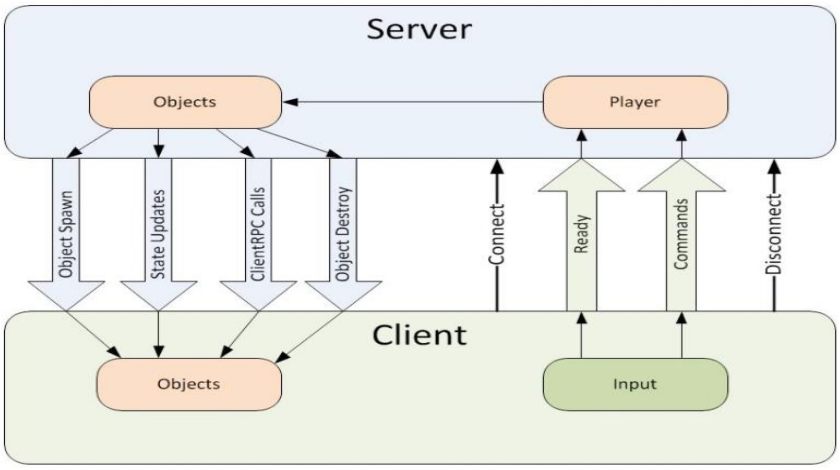


Fig. 3.9 Interacciones del juego [13]

3.5. Interacciones en red

Tras revisar los mecanismos de interacción y las herramientas de red que proporciona *Unity*, se van a explicar a continuación las interacciones en red. Para que un usuario pueda controlar un personaje, debe tener autoridad sobre el objeto. Como se ha explicado en el apartado 3.4, esto se comprueba con las variables booleanas:

```
if (!hasAuthority) { return; }
//ACTUALIZAR MOVIMIENTOS
```

Si el cliente tiene autoridad sobre el objeto podrá actualizar los movimientos, en caso contrario no continúa la ejecución.

Las posiciones de los objetos se actualizan automáticamente añadiendo el bloque *NetworkTransform*. Los objetos que se sincronizan en red deben tener un identificador (*NetworkIdentity*). Además, los estados de la máquina de estados de animaciones se sincronizan con el bloque *NetworkAnimator*.

A continuación, se puede ver el proceso de interacción de un disparo en red (Fig. 3.10) en el juego básico de tanques que se introdujo en el apartado 2.3. Para ello se utilizan comandos y RPCs. En la parte superior de la imagen se pueden ver los mensajes que intercambian el servidor y los clientes. En la parte inferior se representa lo que ve el cliente en su pantalla.

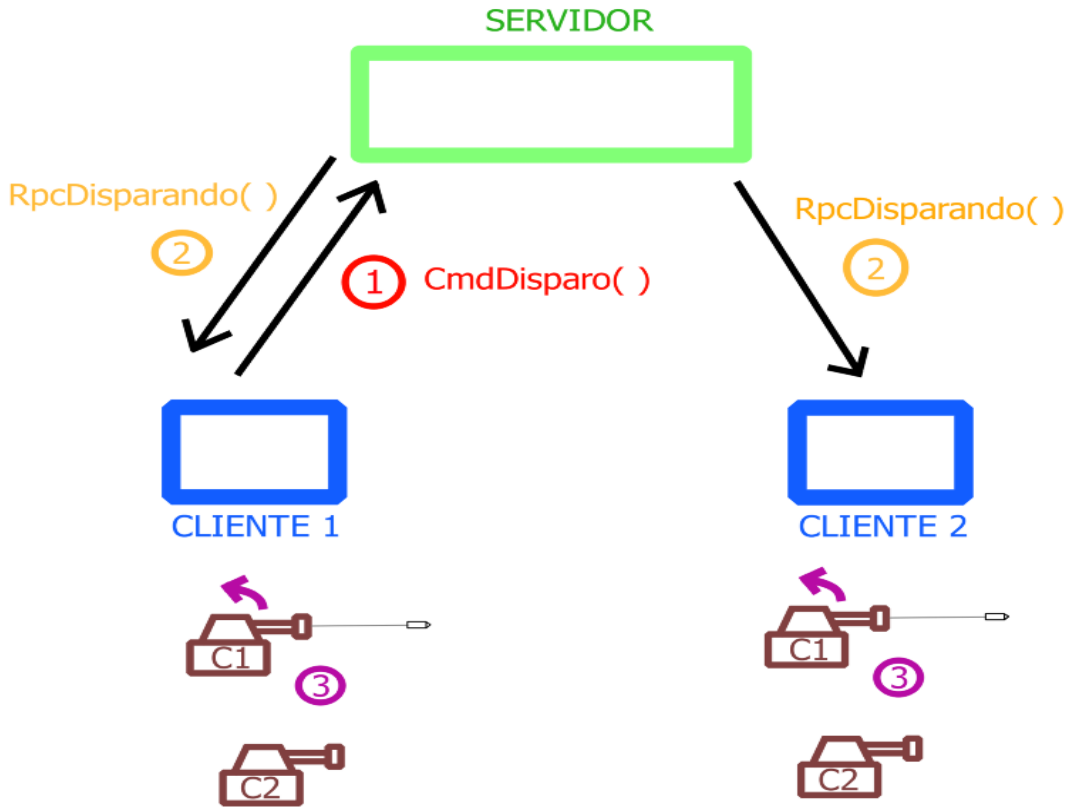


Fig. 3.10 Interacción de un disparo en red

En la figura 3.10 está sucediendo lo siguiente. Primero el cliente 1 pulsa el botón izquierdo del ratón para disparar, entonces ejecuta el comando 1 (*CmdDisparo( )*). El servidor recibe el comando y crea una bala para todos los clientes, el movimiento de la bala no se actualiza en red, sino que los clientes lo simulan. Después, el servidor ejecuta la llamada a procedimiento remoto 2 (*RpcDisparando( )*), todos los clientes reciben este mensaje y ejecutan la animación de disparo 3. En la interacción completa se envían N+1 mensajes, siendo N el número de clientes conectados. El código necesario es el siguiente:

```
[ClientCallback]
private void Update(){

  if (!hasAuthority) { return; } //Continua la ejecución solo para el jugador que posee al
  personaje

  if (Input.GetMouseButtonDown(0))// Botón izquierdo del ratón
        {
            CmdDisparo();
        }

}

[Command]
void CmdDisparo()
{

    GameObject projectile = Instantiate(projectilePrefab, disparo.position, transform.rotation);
    NetworkServer.Spawn(projectile);
    RpcDisparando();

}

[ClientRpc]
void RpcDisparando()
{
   animator.SetTrigger("Dispara");
}
```

Por último, se presentan los objetos que son controlados por el servidor. Estos objetos son las plataformas que se mueven y la barra giratoria. Para que solo el servidor pueda controlar estos objetos, se utiliza la etiqueta [*Server*], como se puede ver en el siguiente código que controla la barra giratoria:

```
float velGiro = 130.0f;

void Update(){

         girar();
}

[Server]
public void girar()
{

        giro.y += velGiro * Time.deltaTime;
        transform.rotation = Quaternion.Euler(90, giro.y, 0);
}
```

Las plataformas del juego se controlan de la misma manera, aunque en ese caso el movimiento es hacia delante y hacia atrás.

3.6. Escena y temática

La aplicación se basa en el juego *Fall Guys*, pero con un personaje más realista. Este juego en red también está hecho con el motor *Unity*. El juego consiste en que un grupo de usuarios aparecen en el inicio y tienen que ir superando obstáculos hasta llegar a la meta. El primero que llega gana.

Fig. 3.11 Captura del juego *Fall Guys*

En el juego desarrollado, el personaje es un arquero que puede disparar flechas. El objetivo es el mismo que en el juego *Fall Guys*, llegar a la meta. La idea es potenciar la interactividad con el juego al tener que competir con otros usuarios por llegar el primero. De esta manera, las condiciones adversas en la red tienen un mayor impacto en el juego. También hay ciertos objetos que son controlados por el servidor, como la barra giratoria y las plataformas que se mueven.

Los objetos controlados por el servidor son los siguientes (Fig. 3.12).

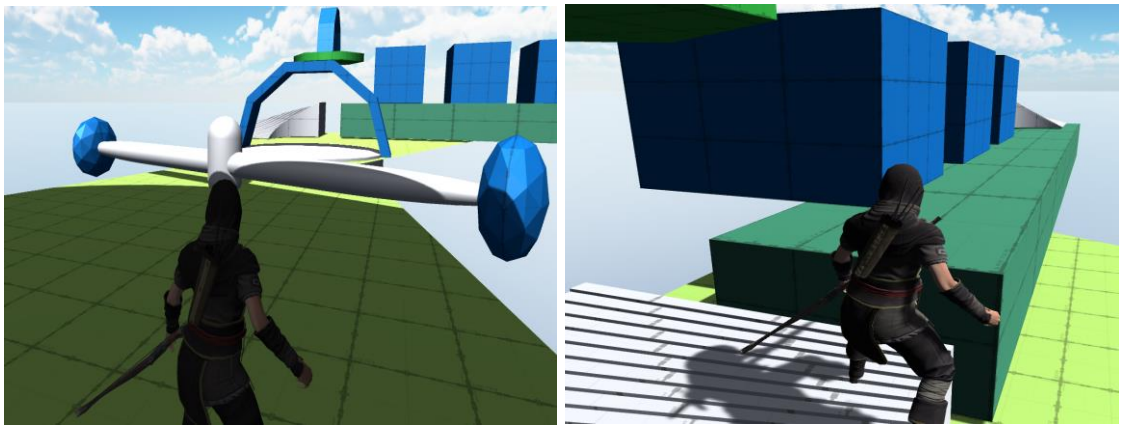


Fig. 3.12 Objetos controlados por el servidor: barra giratoria (izquierda) y plataformas (derecha)

Los objetos de la escena se añaden con la herramienta *ProBuilder*, que permite crear escenas 3D de manera rápida. En la figura 3.13 se puede ver el menú de la herramienta.

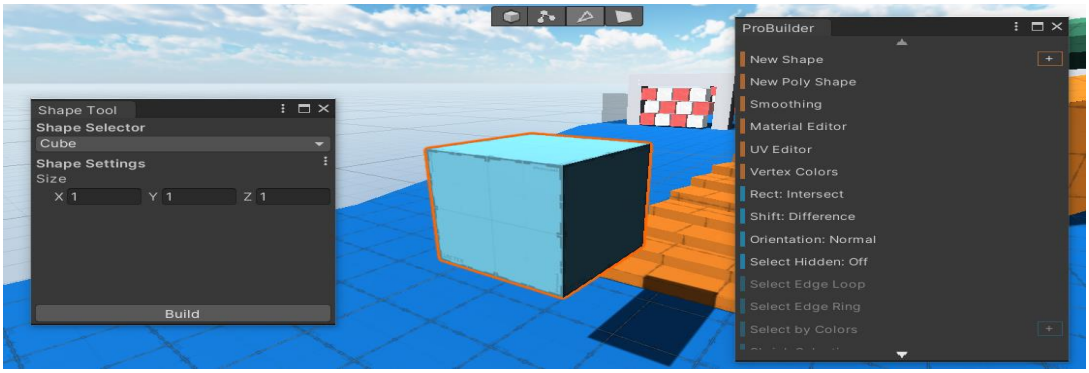


Fig. 3.13 Herramienta *ProBuilder*

Finalmente, se añadió una temática relacionada con los Objetivos de Desarrollo Sostenible (ODS). La idea es concienciar sobre el cambio climático. En el inicio, los usuarios escanean un código QR que los lleva a la página oficial de los ODS (Fig. 3.14).

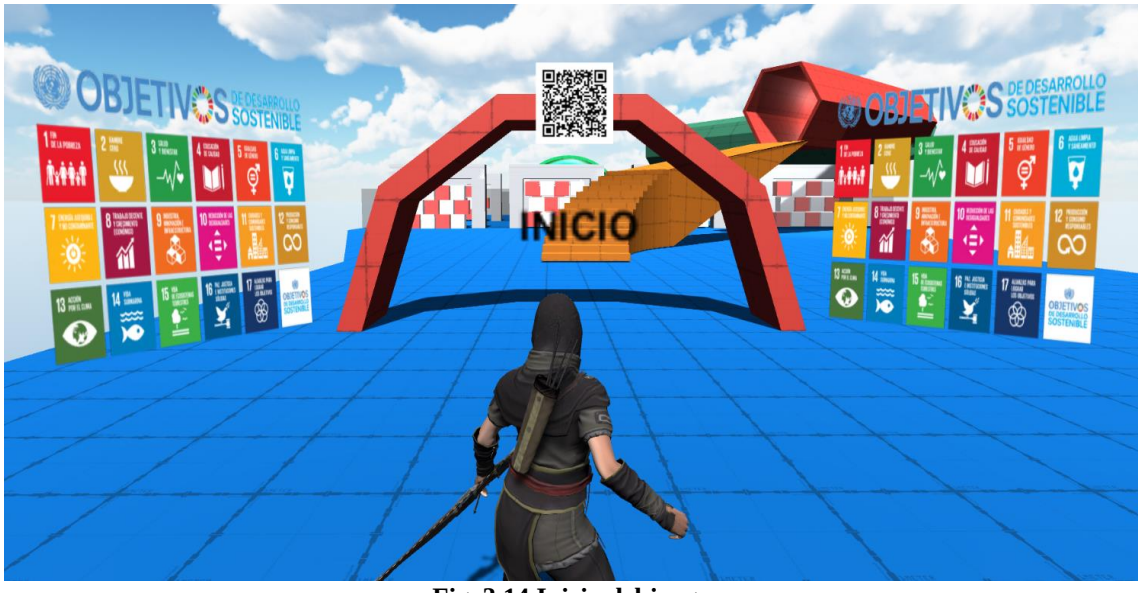


Fig. 3.14 Inicio del juego

Una vez que los usuarios han leído los ODS, se realizan algunas preguntas de tipo verdadero o falso sobre las causas del cambio climático y sus consecuencias. La prueba consiste en 4 puertas, cada puerta tiene encima una de estas preguntas, solo se puede pasar por la puerta que contiene la pregunta verdadera. Si los usuarios aciertan la pregunta sobre los ODS, llegarán más rápido a la meta.

Las preguntas verdaderas son:

- Los gases más contaminantes son: CO_2, CO, metano y ozono.
- Si no se cumplen los ODS, el consumo de energía y las emisiones aumentarán un 50% en 2030 [21].

Las preguntas falsas son:

- El nivel del mar no aumentará en 2100.
- A consecuencia de la subida de temperatura no se extinguirán especies.
- Según los acuerdos, la temperatura media puede aumentar hasta 5º como máximo en 2100 (puede aumentar hasta 2º como máximo [18]).
- La deforestación no contribuye a la subida de la temperatura media.
- No hay relación entre cambio climático y crecimiento de la población.
- El CO_2 es 25 veces más contaminante que el CH_4 (es al revés [7]).

En la figura 3.15 se puede visualizar la prueba de los ODS.

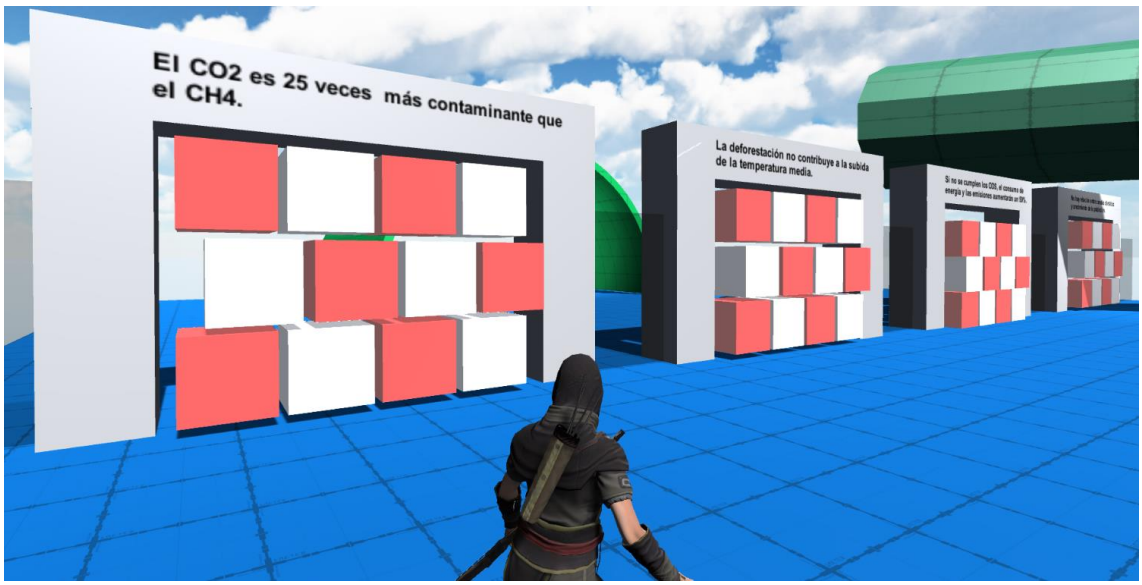


Fig. 3.15 Prueba ODS de verdadero o falso

4. Pruebas de red

Una vez desarrollado el juego, se realizan las pruebas de red. Se comienza caracterizando el tráfico de red generado por la aplicación para distinto número de jugadores y distintas situaciones. Tras ello, se presentan los test MOS que se han realizado a los usuarios para conocer la calidad de la experiencia en función de parámetros de calidad de servicio de la red.

4.1. Caracterización del tráfico

Para caracterizar el tráfico de la aplicación se utiliza el siguiente montaje de laboratorio (Fig. 4.1). Se dispone de un ordenador que actúa como servidor y dos como clientes. En la aplicación, el servidor inicia una partida como *Server Only* y escucha en el puerto 7777. Los clientes inician la partida como *Client* e introducen la dirección IP del servidor (155.210.157.57).

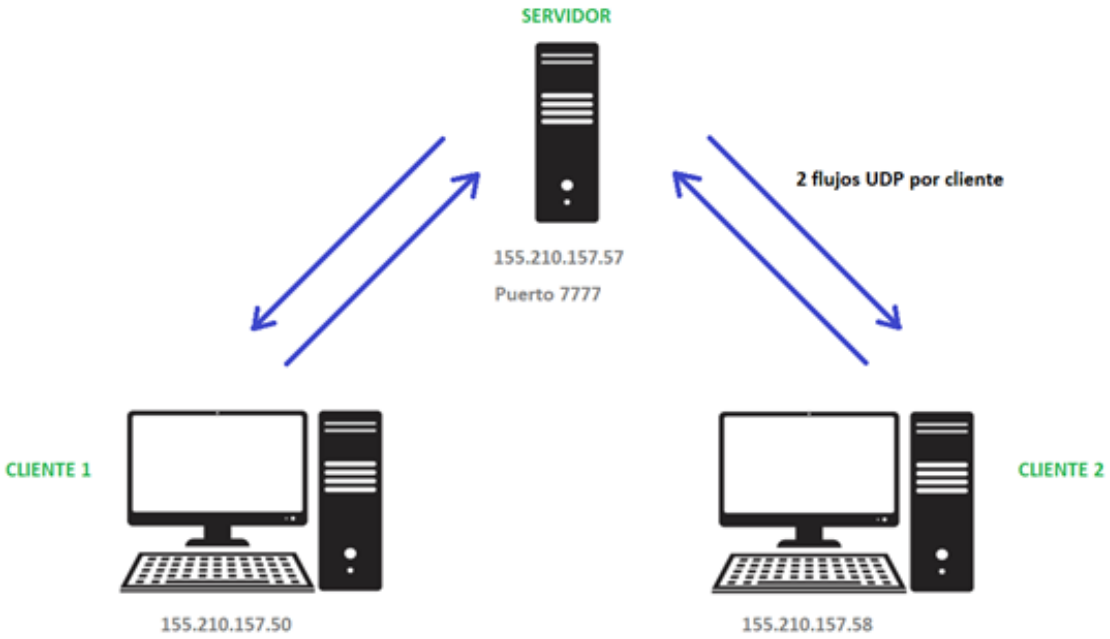


Fig. 4.1 Montaje de laboratorio para caracterizar el tráfico

Como aplicación interactiva en tiempo real, entre cliente y servidor hay 2 flujos de paquetes UDP: el enlace de bajada (de servidor a cliente) y el enlace de subida (de cliente a servidor). Por lo tanto, en este montaje hay 4 flujos en total a analizar por separado en las distintas situaciones: jugadores en reposo y en movimiento. Se utilizará para ello el programa *Wireshark*.

4.1.1. Captura con jugadores en reposo

Se inicia la partida y se captura tráfico durante 3 minutos con los dos jugadores en reposo. Una vez realizada la captura, se importan y se analizan los datos en el programa *Excel*. Primero se puede ver el histograma correspondiente al tamaño de los paquetes para el flujo del cliente al servidor (Fig. 4.2).

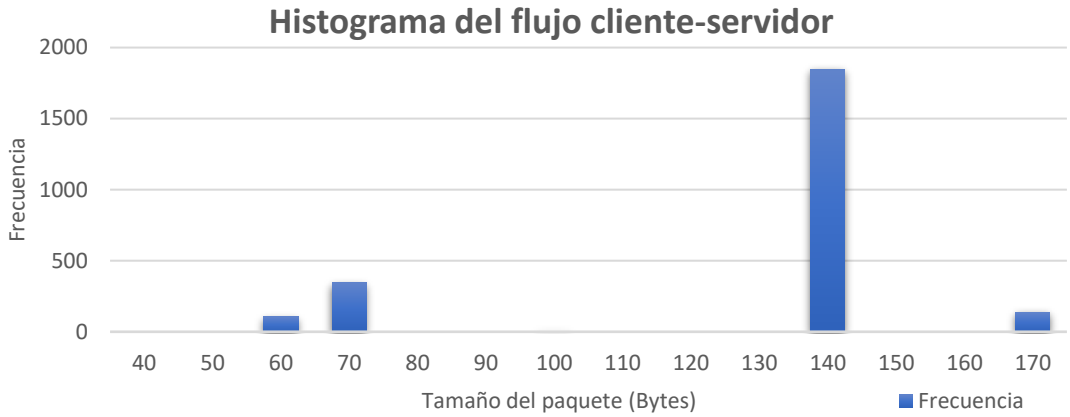


Fig. 4.2 Histograma del enlace de subida con los jugadores en reposo

Como se puede ver en la figura 4.2, hay dos tamaños[1] de paquetes principalmente. Estos se sitúan sobre 70 Bytes y 140 Bytes.

Según la documentación de *Mirror* [13], el bloque encargado de actualizar los movimientos de los jugadores (*NetworkTransform*), envía la posición (x,y,z), la rotación (x,y,z) y la escala (x,y,z) de los jugadores. Estos valores son de tipo *float*, por lo que ocupan 4 Bytes. Si hay que enviar 3 posiciones, 3 rotaciones y 3 escalas de 4 Bytes, esto son 36 Bytes de *payload* como mínimo. Tras descontar las cabeceras conocidas de transporte (8 bytes), red (20 bytes) y enlace (14 bytes), en un paquete de 70 Bytes capturado por *Wireshark* se pueden enviar como mucho 28 Bytes de *payload*. Ello nos lleva a suponer que los paquetes de 70 Bytes se corresponden a los ACK generados por el control de errores ARQ *Selective Repeat* del protocolo KCP. Por lo tanto, los paquetes de 140 Bytes se utilizan para enviar la información sobre la posición de los jugadores. Se envían 2.08 ACKs por segundo frente a 8.47 paquetes de información por segundo, siendo el ancho de banda total de unos 11.4 Kb/s.

A continuación, se presenta el flujo de servidor a cliente (Fig. 4.3). Los paquetes sobre 70 Bytes (ACK) se mantienen respecto al caso anterior. El tamaño de los paquetes de información ha aumentado desde 140 Bytes a 320 Bytes respecto al flujo cliente-servidor, ya que el servidor tiene que enviar la información de los dos clientes. La cadencia de paquetes se mantiene en ambos casos aumentando el ancho de banda total a unos 24.4 Kb/s.

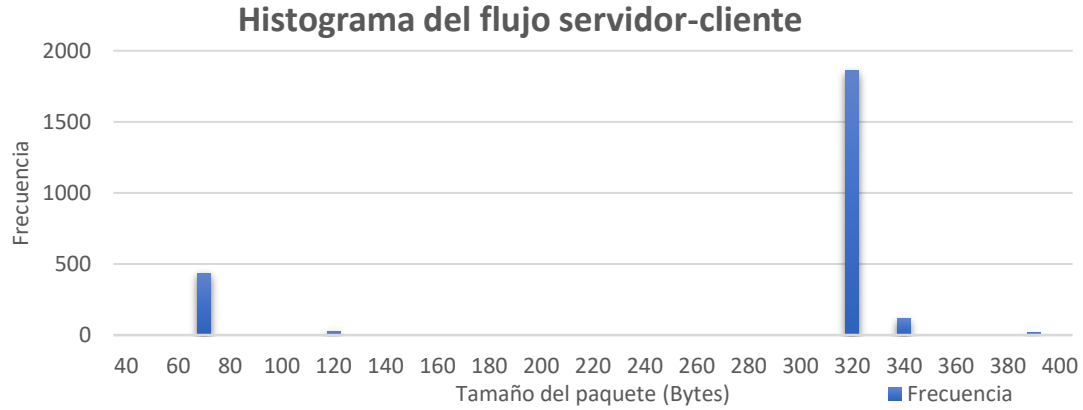


Fig. 4.3 Histograma del enlace de bajada con los jugadores en reposo

[1] Se toma la longitud indicada por Wireshark que únicamente considera 14 bytes de cabecera para el nivel MAC.

En la figura 4.4, se representa el ancho de banda en función del tiempo. Se puede ver que es aproximadamente constante para ambos flujos. Como se ha visto anteriormente, el ancho de banda del flujo servidor-cliente (rojo) es mayor que para el flujo cliente-servidor (verde).

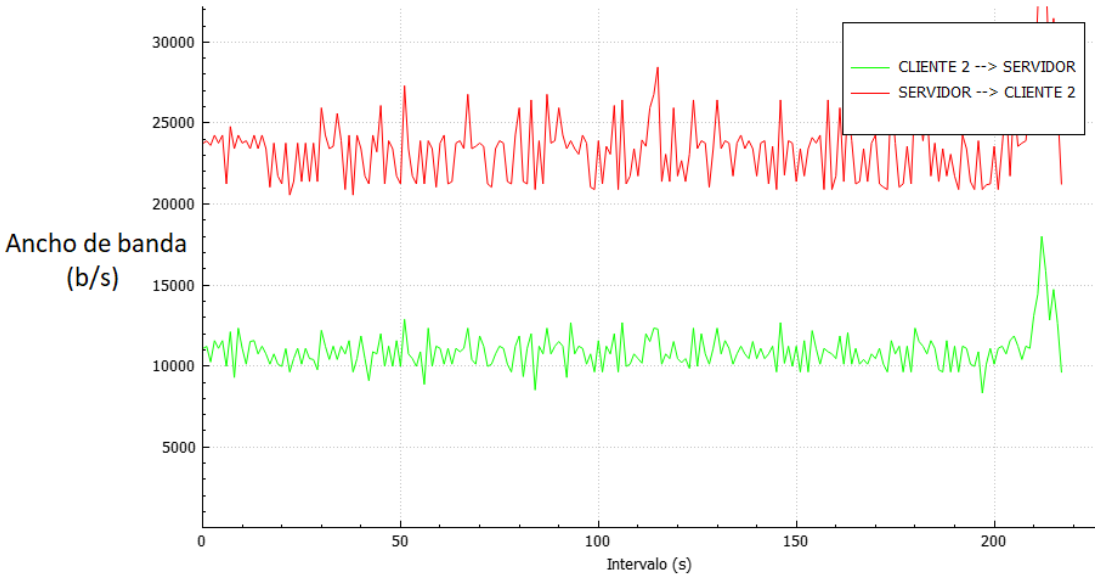


Fig. 4.4 Ancho de banda en función del tiempo para ambos clientes en reposo

4.1.2. Captura con un jugador en movimiento

Tras analizar la situación en reposo, en este subapartado se analiza la situación en la que uno de los clientes (cliente 2) está en movimiento.

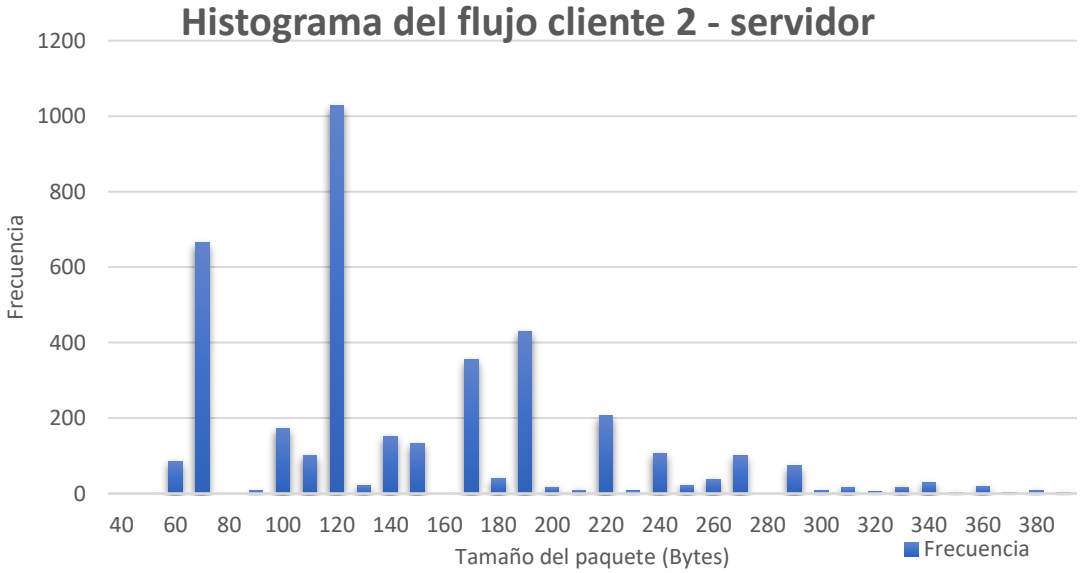


Fig. 4.5 Histograma del enlace de bajada para el jugador en movimiento

En la figura 4.5 también se aprecian los paquetes de ACK (sobre 70 Bytes), en este caso el tamaño de los paquetes de información es muy variable, desde 90 hasta 390 Bytes.

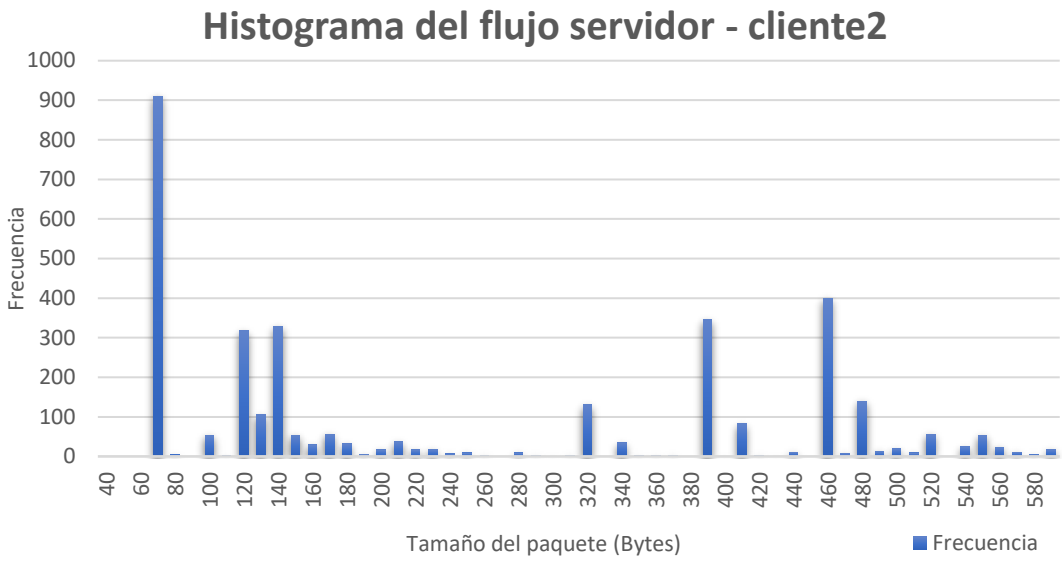


Fig. 4.6 Histograma del enlace de bajada para el jugador en movimiento

En el flujo servidor-cliente (Fig. 4.6), el tamaño de los paquetes es más variable. Se puede ver que el tamaño de los paquetes aumenta respecto al caso sin movimiento (Fig. 4.3). Sin movimiento se situaban en 320 B, mientras que en este caso llegan hasta 590 Bytes.

Si se compara el ancho de banda en función del tiempo para el caso sin movimiento y con movimiento, se aprecia que en el caso con movimiento el ancho de banda es mayor y más variable, frente a un ancho de banda constante en el caso sin movimiento (Fig. 4.7).

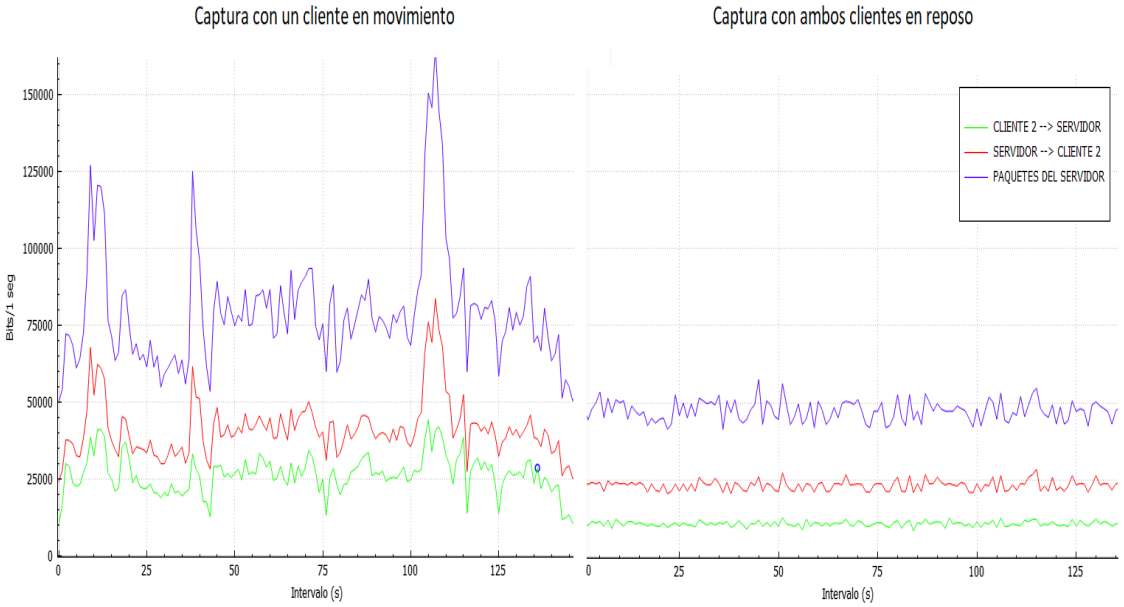


Fig. 4.7 Ancho de banda en función del tiempo para el caso con un cliente en movimiento (izquierda) y ambos clientes en reposo (derecha)

4.1.3. Medidas de tráfico con varios jugadores

En este caso se utiliza un montaje de laboratorio similar al anterior, pero con 4 ordenadores para los clientes. Comenzamos haciendo medidas aumentando paulatinamente el número de clientes en dos condiciones distintas:

- Reposo: Son 4 capturas diferentes, de 1 a 4 clientes. En estas capturas los clientes no realizan ninguna acción.

Tabla 4.1 Capturas con los clientes en reposo

CAPTURA EN REPOSO				
Nº CLIENTES	1	2	3	4
Bajada (S-->C) [ancho de banda, Kbps]	24	24	24	24
Subida (C-->S)	11	11	11	11
Bajada (S-->C) [paquetes por segundo]	11.2	-	-	11.3
Subida (C-->S)	11.3	-	-	11.3
Bajada (S-->C) [tamaño medio del paquete, Bytes]	269	-	-	268
Subida (C-->S)	126	-	-	126

En esta situación (Tabla 4.1), se muestran las medidas referentes a ancho de banda, paquetes por segundo y tamaño medio de paquete, correspondientes a los dos flujos entre cliente y servidor (ambos sentidos). Se puede ver que, en una situación sin movimiento, el ancho de banda no aumenta conforme se incrementa el número de clientes. Esto es debido a que el servidor no tiene que enviar más información de nuevos clientes en el enlace de bajada (coordenadas de los jugadores) ya que la posición de los jugadores no cambia. La cadencia de paquetes se mantiene conforme se incrementa el número de clientes siendo la misma en ambos sentidos. Igualmente, el tamaño medio de los paquetes se mantiene siendo distintos los valores en los dos sentidos.

- Movimiento bajo: Son 4 capturas diferentes, de 1 a 4 clientes. No es una partida real, sino que están realizadas por una sola persona que mueve los 4 teclados a la vez.

Tabla 4.2 Capturas con los clientes en movimiento (poco movimiento)

CAPTURA CON MOVIMIENTO BAJO				
Nº USUARIOS	1	2	3	4
Bajada (S-->C) [ancho de banda, Kbps]	35	42	46	54
Subida (C-->S)	23	27	27	31
Bajada (S-->C) [pps]	23	25.4	26.1	28.4
Subida (C-->S)	23.8	27.4	27.1	29.33
Bajada (S-->C) [tamaño medio del paquete, Bytes]	195	209	222	239
Subida (C-->S)	124	124	129	133

Con poco movimiento (Tabla 4.2), el ancho de banda de bajada es mayor con respecto al caso en reposo y aumenta al incrementar el número de usuarios llegando a duplicar al caso en reposo. Ello se debe a que el servidor va añadiendo la información de la posición

de todos los jugadores. El enlace de subida es mayor con respecto al caso en reposo ya que el cliente envía su posición y acciones al servidor y crece ligeramente con el número de usuarios al crecer el número de ACKs en respuesta a los paquetes enviados desde el servidor. Lógicamente, la cadencia aumenta por el propio movimiento de los usuarios manteniéndose muy parecida en ambos sentidos. Finalmente, vemos que crece ligeramente el tamaño medio los paquetes (apenas en el caso de subida) con valores menores que en reposo debido a la gran dispersión de tamaños existente tal como vimos en los histogramas del subapartado anterior cuando había movimiento.

La figura 4.8 muestra, en las condiciones comentadas, el ancho de banda tanto del enlace de bajada como de subida en función del número y tipo de clientes.

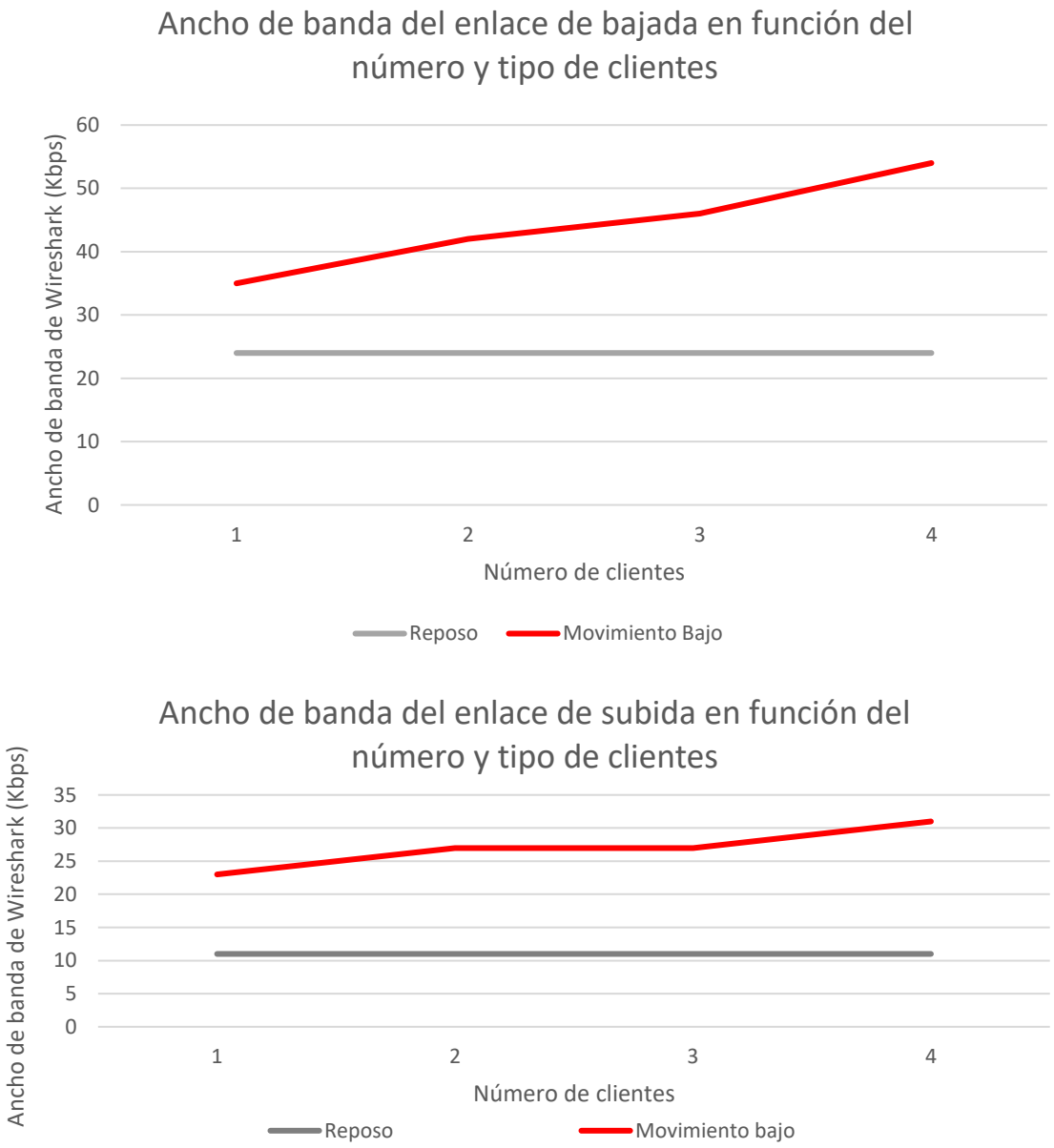


Fig. 4.8 Ancho de banda consumido en el enlace descendente (arriba) y ascendente (abajo) en función del número y tipo de clientes

Consideremos ahora la situación de una partida real entre 4 usuarios distintos:

- Movimiento normal (número de acciones de los clientes).

Tabla 4.3 Captura con los clientes en movimiento (condiciones normales)

PARTIDA REAL DE 4 PERSONAS	
Nº CLIENTES	4
Bajada (S-->C) [ancho de banda, Kbps]	74
Subida (C-->S)	31
Bajada (S-->C) [pps]	27.9
Subida (C-->S)	24.8
Bajada (S-->C) [tamaño medio del paquete, Bytes]	332
Subida (C-->S)	161

En este caso (Tabla 4.3), es significativo el aumento de ancho banda de bajada (triplica al caso en reposo) debido sobre todo al aumento del tamaño medio de paquete. Ello se debe al aumento de acciones de los clientes. Asimismo, debemos ver que la cadencia en ambos sentidos se mantiene bastante similar.

- Movimiento extremo: los clientes realizan muchas más acciones que en una partida normal.

Tabla 4.4 Captura con los clientes en movimiento (condiciones extremas)

PARTIDA REAL DE 4 PERSONAS	
Nº CLIENTES	4
Bajada (S-->C) [ancho de banda, Kbps]	115
Subida (C-->S)	54
Bajada (S-->C) [pps]	37.8
Subida (C-->S)	36.9
Bajada (S-->C) [tamaño medio del paquete, Bytes]	381
Subida (C-->S)	186

En este caso (Tabla 4.4), vuelve a aumentar el ancho de banda de bajada (cuadriplica el caso en reposo) con una cadencia de paquetes superior al caso de movimiento normal (Tabla 4.3). Esto es debido a que los clientes realizan muchas acciones (disparos y movimientos), y esto requiere de un mayor intercambio de paquetes. Esta cadencia de paquetes junto al aumento significativo del tamaño medio de paquete lleva al aumento del ancho de banda mencionado. Como en el caso anterior la cadencia en ambos sentidos es similar llevando a un aumento del ancho de banda de subida.

Considerando todas las condiciones expuestas, se aprecia un crecimiento paulatino del ancho de banda de bajada cuando hay movimiento conforme se aumenta el número de clientes (pendiente en figura 4.8) y un aumento muy significativo cuando se pasa de estado de reposo a movimiento extremo (multiplicando por 4) con estadios intermedios de movimiento bajo (por 2) y movimiento normal (por 3). Estas ideas deberían tenerse muy en cuenta a la hora de diseñar el ancho de banda que debería tener el servidor en

función del número de clientes a los que dar servicio. En el caso de los clientes, normalmente situados en lugares distintos cada uno, el ancho de banda no es un parámetro importante a considerar. Su crecimiento paulatino se debe más al ancho de banda asociado a los ACK. Más importantes son las condiciones de red, como se verá en el siguiente apartado.

Finalmente es necesario tener en cuenta que el ancho de banda medido se ha contabilizado en una LAN Ethernet. Sería más interesante conocer el ancho de banda consumido a nivel IP y tener así valores independientes de la tecnología subyacente. Para ello, y teniendo en cuenta la pila de protocolos empleada (figura 4.9), habría que descontar el ancho de banda equivalente a los 14 bytes de cabecera Ethernet. Esto lo haríamos multiplicando esos bytes por el número medio de paquetes por segundo.

$$BW_{1\ CLIENTE\ NIVEL\ IP} = BW_{1\ CLIENTE\ DE\ WIRESHARK} - (cabeceras\ Ethernet * pps) \qquad \textbf{(Ec. 4.1)}$$

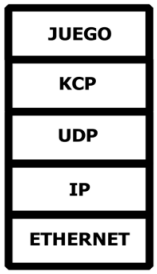


Fig. 4.9 Pila de la aplicación

Así, el ancho de banda de bajada (a nivel IP) con un único cliente en caso de movimiento bajo es:

$BW_{1\ CLIENTE\ BAJADA\ NIVEL\ IP}$ = 35 Kbps – 14 Bytes * 8 bits/Byte * 23 pps = 32,4 Kbps.

De la misma manera, con 4 clientes, el ancho de banda de bajada por cada flujo en movimiento extremo será:

$BW_{4\ CLIENTES\ BAJADA\ NIVEL\ IP}$ = 115 Kbps – 14 Bytes * 8 bits/Byte * 37,8 pps = 110,7 Kbps bps.

4.2. QoE: Pruebas de red a usuarios

La experiencia del usuario al utilizar una aplicación es un factor importante. Si un usuario no se siente cómodo utilizando una aplicación, es probable que deje de utilizarla. En un juego on-line esta experiencia puede verse afectada por condiciones adversas de la red. Los parámetros de red que se estudian en estas pruebas son: latencia, variación de latencia (*jitter*), pérdida de paquetes constante y pérdida de paquetes a ráfagas. La Unión Internacional de Telecomunicaciones (ITU-T) ofrece recomendaciones para evaluar la calidad de experiencia del usuario o QoE (*Quality of Experience*) mediante un test MOS (*Mean Opinion Score*), por ejemplo, la recomendación P.800.1[12]. Estas recomendaciones están orientadas a evaluar la calidad del audio y vídeo principalmente.

Para evaluar la QoE de esta aplicación se utiliza un test MOS. Esta es una evaluación subjetiva de la aplicación. Aunque serían necesarias una gran cantidad de voluntarios para

que el test fuera representativo, se realiza una primera aproximación con un grupo de N=7 personas (4 hombres y 3 mujeres). Todos ellos eran estudiantes de entre 17-23 años. Algunas pruebas fueron realizadas individualmente y otras en grupo.

En un test MOS se utiliza la siguiente ecuación:

$$MOS = \frac{\sum_{n=1}^{N} R_n}{N} \qquad \textbf{(Ec. 4.2)}$$

Siendo R las calificaciones de los usuarios y N el número de usuarios. Las calificaciones van desde 1 (malo) hasta 5 (excelente).

Se utiliza un montaje de laboratorio similar a los anteriores. En la figura 4.10 se puede ver el procedimiento para que el cliente 1 realice la prueba. Primero se aplica la condición de red en el enlace ascendente con el programa *Network Emulator Client*. Una vez aplicada, el cliente 1 realiza acciones en el juego mirando en la pantalla del cliente 2. De esta manera, puede ver el efecto que producen estas condiciones de red y califica el funcionamiento del juego del 1 al 5.

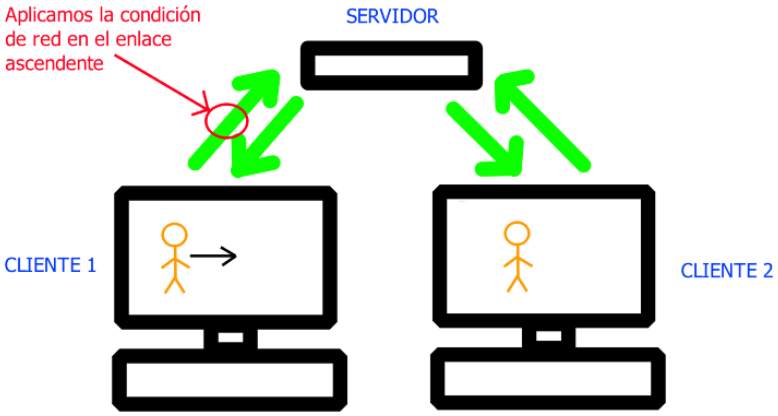


Fig. 4.10 Evaluación subjetiva para el cliente 1 con un test MOS

En los siguientes subapartados se presentan las 5 pruebas realizadas: latencia, latencia variable (*jitter*), pérdida de paquetes constante, pérdida de paquetes a ráfagas y pérdida de paquetes con latencia.

4.2.1. Prueba 1: latencia constante

En la primera prueba se introducen retardos constantes desde 100 a 500 milisegundos. Una vez realizada la prueba, se presenta la opinión media del usuario en función de la latencia en la figura 4.11. Se puede ver que la opinión media se mantiene por encima de 4 (buena calidad) hasta 200 ms de latencia. A partir de 300 ms la opinión media comienza a bajar de 3, lo que quiere decir que la calidad no es aceptable.

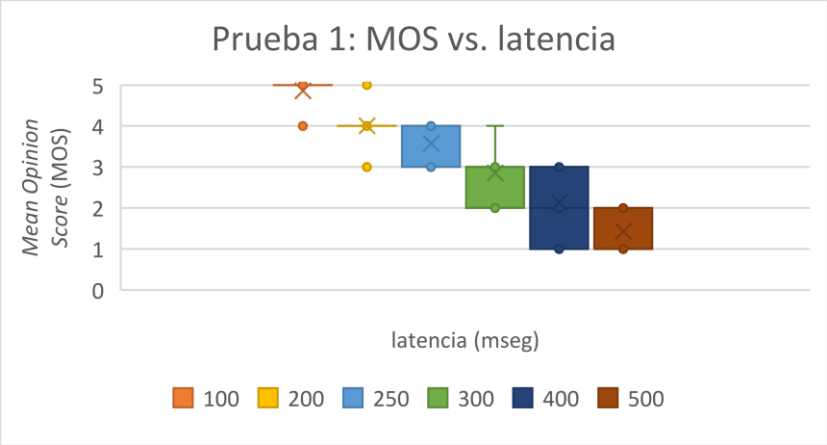


Fig. 4.11

4.2.2. Prueba 2: latencia variable (*jitter*)

En la segunda prueba se aplica un retardo variable o *jitter*. Se obtiene una calidad media de 4 para el intervalo de jitter entre 100 y 200 ms. En este caso, la calificación comienza a bajar de 3 a partir de 200-300 ms, mientras que en la prueba 1 eran necesarios 300 ms para que la opinión media bajara de 3 puntos.

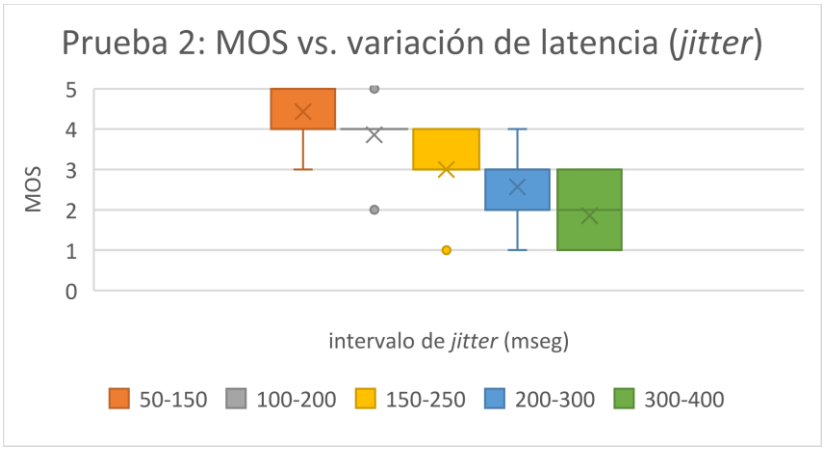


Fig. 4.12

4.2.3. Prueba 3: pérdida de paquetes constante

En esta prueba se aplica una pérdida de paquetes constante con una probabilidad determinada. Se mantiene una buena calidad (puntuación igual a 4) con una pérdida del 20% de los paquetes (uno de cada cinco). En la figura 4.13 se puede ver que se mantiene una calidad aceptable (puntuación igual a 3) hasta un 33% de pérdida de paquetes. Esto se debe a que el juego puede compensar estas pérdidas gracias a su control de errores y otros mecanismos de la aplicación que predicen la siguiente posición del jugador a partir de las anteriores.

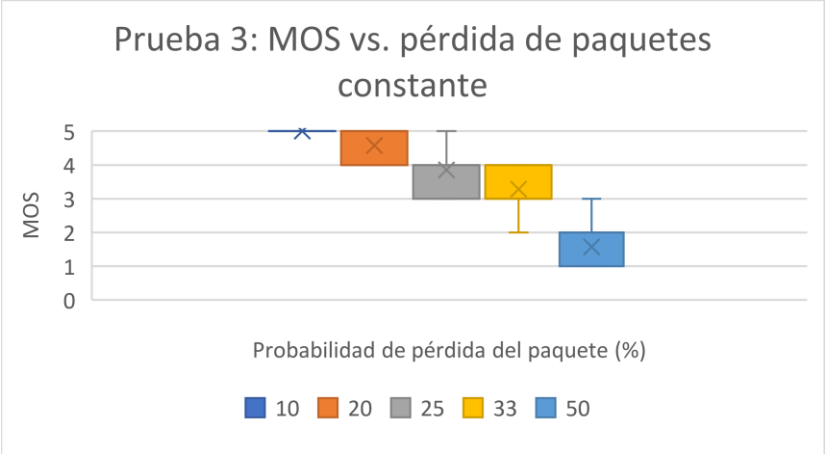


Fig. 4.13

4.2.4. Prueba 4: pérdida de paquetes a ráfagas

En la cuarta prueba se introduce una pérdida de paquetes a ráfagas con una probabilidad de aparición determinada. La duración de la ráfaga está uniformemente distribuida entre uno y tres paquetes. En la figura 4.14 se puede ver que con una probabilidad del 20%, algunas calificaciones llegan hasta 2 (calidad pobre), mientras que en la prueba 3 las calificaciones se situaban entre 4 y 5, es decir, entre bueno y excelente (Fig. 4.13). Cuando se pierden varios paquetes seguidos, el control de errores y la predicción de movimiento no siempre pueden compensar estas pérdidas.

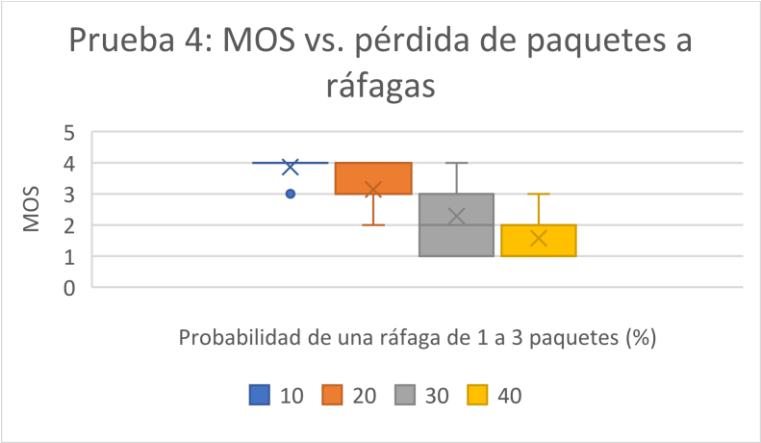


Fig. 4.14

4.2.5. Prueba 5: pérdida de paquetes y latencia constantes

Por último, se pretende simular una situación más realista que combine la latencia y la pérdida de paquetes. Para una pérdida del 20% de los paquetes combinada con una latencia de 150 ms (Fig. 4.15) se obtienen calificaciones entre 1 y 3. Estas condiciones de red, aplicadas por separado, no afectaban tanto a la calidad del juego ya que se obtuvieron calificaciones entre 4 y 5 (Figuras 4.11 y 4.13)

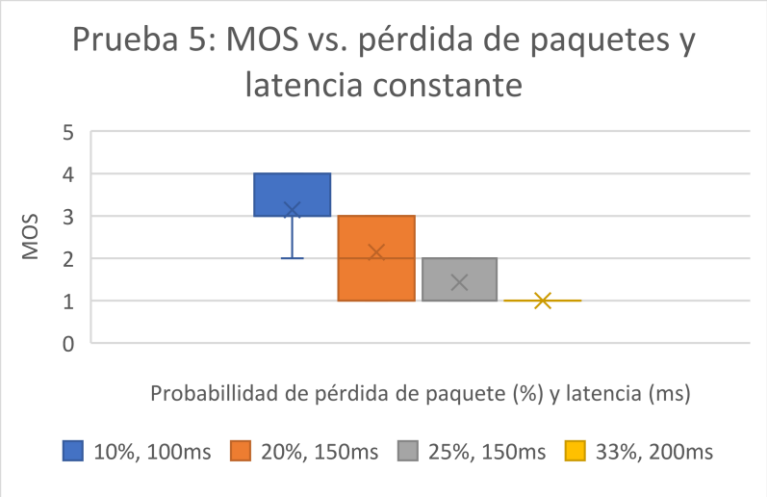


Fig. 4.15

4.3. Resultados de las pruebas

Como se ha visto en el apartado anterior, es posible jugar con calidad hasta un valor de latencia de 200 ms. En cuanto al *jitter*, el intervalo de latencia puede estar entre 100 y 200 ms. Para situar geográficamente un servidor hay que tener en cuenta estos valores y dejar un margen. La distancia máxima del servidor a los usuarios está limitada por estos retardos.

Según los usuarios, sin considerar la prueba 5, la prueba que más afecta a la jugabilidad es la pérdida de paquetes a ráfagas. Los resultados se presentan en la figura 4.16.

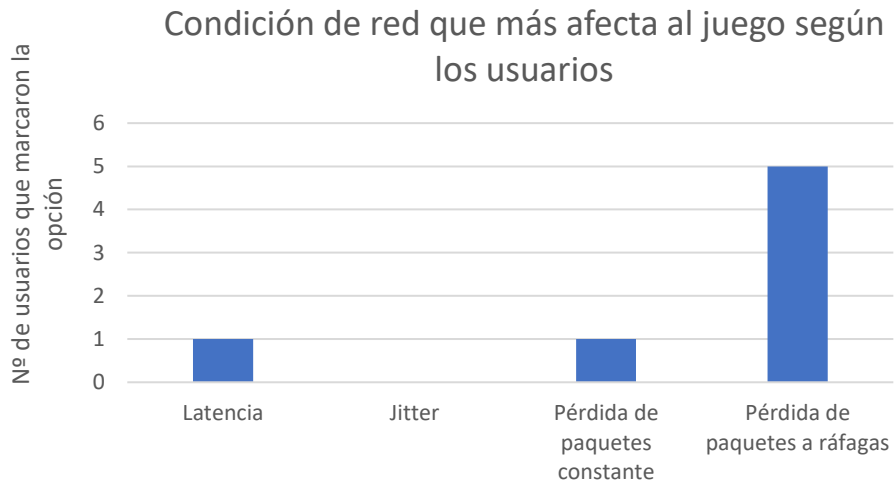


Fig. 4.16

Otra consideración de las pruebas realizadas es que las personas sin experiencia en videojuegos asignaron calificaciones más altas que las personas con experiencia.

5. Conclusiones

Tras terminar el trabajo, se presentan las conclusiones:

- El motor de videojuegos *Unity 3D* y la API de red *Mirror* son una muy buena opción para realizar aplicaciones audiovisuales en red.

- El transporte de red KCP se comporta bien en aplicaciones muy interactivas. En este caso se ha utilizado para un juego FPS, pero podría utilizarse también en otro tipo de aplicaciones con un tráfico y necesidades de red similares.

- En una aplicación de este tipo, se distinguen dos flujos entre el cliente y el servidor: el flujo descendente (servidor a cliente) y el flujo ascendente (cliente a servidor). En el flujo descendente, el servidor envía la información de todos los jugadores, mientras que, en el flujo ascendente, el cliente envía su información al servidor (posición y acciones). Al aumentar el número de usuarios, el ancho de banda del flujo descendente crece más rápido que en el flujo ascendente ya que el servidor tiene que ir añadiendo la información de los nuevos clientes. El aumento del ancho de banda se debe a un incremento del tamaño medio del paquete.

- El ancho de banda puede crecer de dos maneras: al aumentar el número de usuarios y según el movimiento de los usuarios. Con el aumento del número de usuarios el ancho de banda crece de forma paulatina, mientras que al pasar de una situación en reposo a un movimiento extremo el ancho de banda de bajada aumenta en un factor 4 aproximadamente.

- Los resultados de las pruebas muestran que es posible jugar con calidad con una latencia de 200 ms, un *jitter* de entre 100 a 200 ms o una pérdida del 20% de los paquetes. La pérdida de paquetes a ráfagas es el parámetro de red que más afecta a la jugabilidad.

Como trabajo futuro se propone:

- Añadir más interacciones en red. Por ejemplo, enemigos con inteligencia artificial controlados por el servidor. O profundizar más en el tema de las variables sincronizadas en red.

- Añadir streaming de audio para que los usuarios puedan hablar entre ellos.

- Realizar un mayor número de medidas de partidas reales para obtener las ecuaciones que permitan estimar el ancho de banda necesario en el servidor en función del número de usuarios.

Los problemas encontrados fueron:

- Para realizar los test MOS fueron necesarios ordenadores con tarjeta gráfica dedicada, ya que en los ordenadores sin tarjeta gráfica dedicada la GPU alcanzaba el 100% de uso y el rendimiento del juego se veía afectado.

- Tener cuidado con las herramientas de red del sistema operativo *Windows* ya que su configuración puede afectar a las pruebas de red a realizar. Ejemplo de ello es el *Firewall* de *Windows* que puede llegar a filtrar paquetes e incluso rechazar conexiones de red.

Sobre la organización temporal del trabajo, se han conseguido los objetivos iniciales en el plazo previsto. En la siguiente figura se puede ver el cronograma del trabajo.

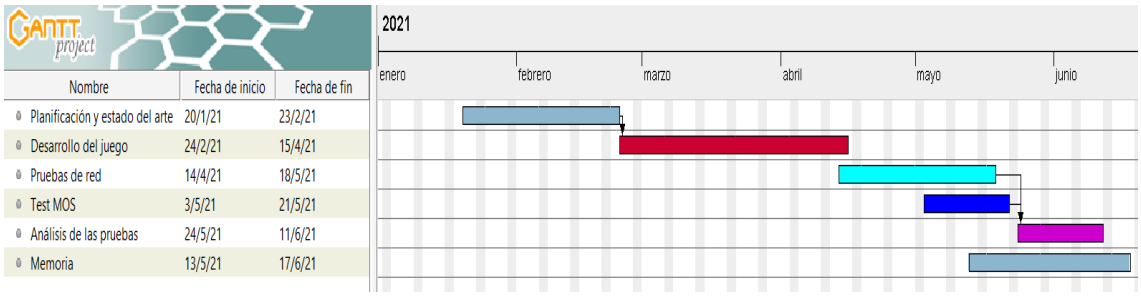


Fig. 5.1 Cronograma del trabajo

Anexo A: Código desarrollado

En este anexo se presenta el código desarrollado para la aplicación.

A.1. Movimiento del personaje

Este es el código que permite al usuario controlar el personaje.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Mirror;

Unity Script | 0 references
public class PlayerMovement : NetworkBehaviour
{
    [Header("References")]
    [SerializeField] private CharacterController controller = null;
    [SerializeField] private Animator animator = null;

    [Header("Settings")]
    [SerializeField] private Vector3 moveVector;
    private float gravedad = -45.0f;// Es más realista que con g = -9.8f
    float fuerzaSalto = 1.5f;

    [SerializeField] private float mouseX;
    private float mouseY;
    private float VelocidadRotacion = 3.0F;
    private float Velocidad = 8.0f;
    [SerializeField] public Vector3 velocidadCaida;

    [SerializeField] public Transform sueloCheck;
    [SerializeField] public Transform disparo;
    public float distanciaSuelo = 0.4f;
    public LayerMask sueloMask;
    [SerializeField] private bool isGrounded;
    [SerializeField] private GameObject projectilePrefab = null;
    private bool apunta;
    private Vector3 respawn;
    private GameObject respawn1;


    [ClientCallback]
    Unity Message | 0 references
    private void Start()
    {
        if (!hasAuthority) { return; }// Continua la ejecución solo para el jugador que posee al personaje

        // Mueve la cámara principal al personaje
        Cursor.visible = false;
        Cursor.lockState = CursorLockMode.Locked;// Cursor bloqueado e invisible
        Camera.main.transform.position = this.transform.position - this.transform.forward * 2.2f +
        this.transform.up * 2;// Ajusta la canara al personaje

        Camera.main.transform.LookAt(this.transform.position);
        Camera.main.transform.parent = this.transform;

        respawn = transform.position; // Punto de reaparición del personaje
        respawn1 = GameObject.Find("Respawn1");

    }


    [ClientCallback]
    Unity Message | 0 references
    private void Update()
    {
        if (!hasAuthority) { return; } //Continua la ejecución solo para el jugador que posee al personaje

        // Movimiento de la cámara y del personaje con el ratón
        mouseX += Input.GetAxis("Mouse X")*VelocidadRotacion;// Eje vertical (Vista)
        mouseY -= Input.GetAxis("Mouse Y")*VelocidadRotacion;// Eje Horizontal
        mouseY = Mathf.Clamp(mouseY, -20, 60);

        transform.rotation = Quaternion.Euler(0, mouseX, 0); // Se aplica la rotación al personaje
        Camera.main.transform.rotation = Quaternion.Euler(mouseY, mouseX, 0); // Se aplica la rotación a la cámara



        // Se aplica la gravedad y se controla el salto
        isGrounded = Physics.CheckSphere(sueloCheck.position, distanciaSuelo, sueloMask);
        animator.SetBool("Suelo", Physics.CheckSphere(sueloCheck.position, 1.0f, sueloMask));

        if (isGrounded && velocidadCaida.y < 0)
        {
            velocidadCaida.y = -2f;
        }

        velocidadCaida.y += gravedad * Time.deltaTime;
        controller.Move(velocidadCaida * Time.deltaTime);// (Ecuación) variación posición = 1/2*g*t^2

        if (Input.GetButtonDown("Jump") && isGrounded && animator.GetCurrentAnimatorStateInfo(0).IsName("IDLE (Reposo)"))
        {
            animator.SetBool("Salto", true);
            velocidadCaida.y = Mathf.Sqrt(fuerzaSalto * -2f * gravedad);// (Ecuación) v = sqrt(h *-2*g)
        }
```

```
        else if (Input.GetButtonDown("Jump") && isGrounded && animator.GetCurrentAnimatorStateInfo(0).IsName("Movimiento"))
        {
            animator.SetBool("SaltoCorriendo", true);
            velocidadCaida.y = Mathf.Sqrt(fuerzaSalto * -2f * gravedad);// (Ecuación) v = sqrt(h *-2*g)
        }
        else
        {
            animator.SetBool("Salto", false);
            animator.SetBool("SaltoCorriendo", false);
        }

        // Controles del jugador (Actualizar posición)
        var movement = new Vector3();//Vector de posiciones

        if (Input.GetKey(KeyCode.W) ) { movement.z += 1;  }
        if (Input.GetKey(KeyCode.S)) { movement.z -= 1; }
        if (Input.GetKey(KeyCode.A)) { movement.x -= 1; }
        if (Input.GetKey(KeyCode.D)) { movement.x += 1; }

        Vector3 Movimiento = new Vector3(movement.x,0, movement.z)*Time.deltaTime*Velocidad;
        transform.Translate(Movimiento, Space.Self);// Se aplica el movimiento

        // Animación de apuntar con el arco
        apunta = false;
        if (Input.GetMouseButton(1))// Botón derecho del ratón
        {
            apunta = true;
            Velocidad = 4.0f;
        }
        else
        {
            Velocidad = 8.0f;
        }
        animator.SetBool("Apuntando", apunta);

        // Disparo
        if (Input.GetMouseButtonDown(0))// Botón izquierdo ratón para disparar
        {
            CmdDisparo();
        }

        //Controlar la animacion de movimiento del personaje
        animator.SetFloat("VelX", movement.x);
        animator.SetFloat("VelZ", movement.z);
        animator.SetBool("Caminando", (movement.x != 0.0f) || (movement.z != 0.0f) );

        // Comprobar caida fuera del mapa
        if(transform.position.y < -55.0f)
        {
            transform.position = respawn;// Lleva al personaje al punto de reaparición
        }

        // Actualizar punto de reaparición
        if (Vector3.Distance(transform.position, respawn1.transform.position) < 5.0f)
        {
            respawn = respawn1.transform.position;
        }
    }

    [Command]
    1 reference
    void CmdDisparo()
    {
        GameObject projectile = Instantiate(projectilePrefab, disparo.position, transform.rotation);
        NetworkServer.Spawn(projectile);
        RpcDisparando();
    }

    [ClientRpc]
    1 reference
    void RpcDisparando()
    {
        animator.SetTrigger("Dispara");
    }
}
```

A.2. Control de la barra giratoria

Con este código el servidor controla la barra giratoria.

```
public class movGiro : NetworkBehaviour
{
    Vector3 giro = new Vector3(0, 0, 0);
    float velGiro = 130.0f;

    Unity Message | 0 references
    void Update()
    {
        girar();
    }


    [Server]
    1 reference
    public void girar()
    {
        giro.y += velGiro * Time.deltaTime;
        transform.rotation = Quaternion.Euler(90, giro.y, 0);
    }
}
```

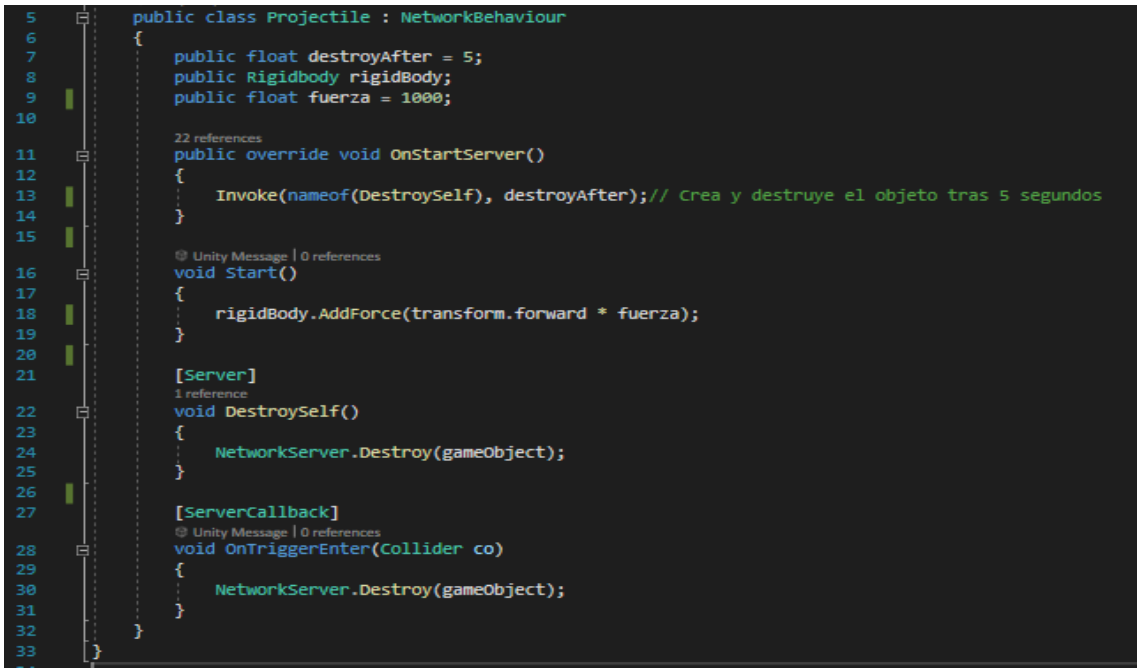
A.3. Control de las plataformas

Este código permite al servidor controlar las plataformas del juego.

```
public class MovimentoCubo : MonoBehaviour
{
    Vector3 posInicial;
    float velocidad = 0.012f;
    bool adelante = true;//Movimiento hacia delante o hacia atras del cubo

    Unity Message | 0 references
    void Start()
    {
        posInicial = transform.position;
    }


    Unity Message | 0 references
    void Update()
    {
        movimiento();
    }


    [Server]
    1 reference
    public void movimiento() { //El objeto es controlado por el servidor
        if(posInicial.z - transform.position.z > 3.0f)
        {
            adelante = false;
        }
        else if (posInicial.z - transform.position.z < 0.0f)
        {
            adelante = true;
        }
        if (adelante)
        {
            transform.Translate(0,0,transform.position.z * Time.deltaTime * -1.0f*Velocidad);
        }
        else
        {
            transform.Translate(0,0,transform.position.z * Time.deltaTime * Velocidad);
        }
    }
}
```

A.3. Control de las flechas

Con el siguiente código se controlan las flechas que dispara el jugador. Crea la flecha y le aplica un movimiento. Tras 5 segundos se elimina el objeto.



```
public class Projectile : NetworkBehaviour
{
    public float destroyAfter = 5;
    public Rigidbody rigidBody;
    public float fuerza = 1000;

    22 references
    public override void OnStartServer()
    {
        Invoke(nameof(DestroySelf), destroyAfter);// Crea y destruye el objeto tras 5 segundos
    }

    Unity Message | 0 references
    void Start()
    {
        rigidBody.AddForce(transform.forward * fuerza);
    }

    [Server]
    1 reference
    void DestroySelf()
    {
        NetworkServer.Destroy(gameObject);
    }

    [ServerCallback]
    Unity Message | 0 references
    void OnTriggerEnter(Collider co)
    {
        NetworkServer.Destroy(gameObject);
    }
}
}
```

Referencias

1. Adobe. *Mixamo animate 3D characters for games, film, and more.* https://www.mixamo.com/#/
2. AEVI. (2021). La industria de los videojuegos en España., 84. http://www.aevi.org.es/web/wp-content/uploads/2021/04/AEVI_Anuario_2020.pdf
3. Alarcón Cano, S. (2020). Videojuego de lucha uno contra uno en red y con inteligencia artificial adaptativa . Universidad de Zaragoza (EINA)). , 115.
4. *Bloqueo del cardán.* (2019). Última consulta: 11 de junio 2021 https://es.wikipedia.org/w/index.php?title=Bloqueo_del_card%C3%A1n&oldid=119169170
5. *CRYENGINE | the complete solution for next generation game development by crytek.* Última consulta: 18 de mayo de 2021 https://www.cryengine.com/
6. Europa Press. (2020). *La industria de los videojuegos alcanza los 120.100 millones de dólares en 2019.* Última consulta: 18 de mayo de 2021 https://www.europapress.es/portaltic/videojuegos/noticia-industria-videojuegos-alcanza-120100-millones-dolares-2019-20200102170428.html
7. Gases de efecto invernadero. (2014). https://obccd.org/informacion-basica-2/gases-de-efecto-invernadero-co2e-co2-y-carbono/

8. *Introducing KCP: A new low-latency, secure network stack.* (2019).

   Última consulta: 27 de mayo de 2021

   https://www.improbable.io/blog/kcp-a-new-low-latency-secure-network-stack

9. *Introduction | photon engine.*

   Última consulta: 29 de marzo de 2021

   https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro

10. *KCP - A fast and reliable ARQ protocol.*

    Última consulta: 28 de mayo de 2021

    https://github.com/skywind3000/kcp/blob/master/README.en.md

11. Marcén Altarriba, R. (2017). Creación de un videojuego de carreras en 3D. Universidad de Zaragoza (EINA). , 66.

12. *Mean opinion score.* (2020).

    Última consulta: 15 de junio de 2021

    https://en.wikipedia.org/w/index.php?title=Mean_opinion_score&oldid=997226082

13. Mirror. *Documentation: Transports.*

    Última consulta: 27 de mayo de 2021

    https://mirror-networking.gitbook.io/docs/transports

14. *Mirror networking – open source networking for unity.*

    Última consulta: 29 de marzo de 2021

    https://mirror-networking.com/

15. *Motor de videojuego.* (2020).

    Última consulta:18 de mayo de 2021,

    https://es.wikipedia.org/w/index.php?title=Motor_de_videojuego&oldid=127730481

16. Pachi (2019). *Unity 2019: Importar y animar un personaje 3D* [Video].

https://www.youtube.com/watch?v=Ay_oy6GXC-s&t=6s

17. *Plataforma de desarrollo en tiempo real de unity | motor de VR y AR en 3D y 2D.*

Última consulta: 25 de mayo de 2021

https://unity.com/es

18. Rovira, M. (2018). *De 2 °C a 1,5 °C, medio grado crucial.*

Última consulta: 17 de junio de 2021

https://www.lavanguardia.com/vida/junior-report/20181210/453485814370/ipcc-calentamiento-global-cambio-climatico-temperatura-2-grados-15-grados.html

19. Thirslund, A (2019). *FIRST PERSON MOVEMENT in unity - FPS controller.* [Video]

https://www.youtube.com/watch?v=_QajrabyTJc

20. Turnage, P. (2020). *Latency of reliable streams.*

Última consulta: 27 de mayo de 2021

https://paytonturnage.com/writing/latency-of-reliable-streams/

21. UNESCO. *Hecho 19: La demanda energética.*

http://www.unesco.org/new/es/natural-sciences/environment/water/wwap/facts-and-figures/all-facts-wwdr3/fact-19-energy-needs/

22. *Unity - manual: Multiplayer and networking.*

Última consulta: May 18, 2021

https://docs.unity3d.com/Manual/UNet.html

23. *Unreal engine | the most powerful real-time 3D creation platform.*

Última consulta: 29 de marzo de 2021

https://www.unrealengine.com/en-US/

24. Zhangyuan. (2015). *Reliable UDP bench mark.*

Última consulta: 27 de mayo de 2021

https://github.com/libinzhangyuan/reliable_udp_bench_mark/blob/master/bench_mark.md